Universität Regensburg

Wissenschaftliches Rechnen auf Grafikkarten

Universität Regensburg

Thomas Karl

Sommersemester 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Codebeispiele

Glossar

API application programming interface, Programmierschnittstelle. 20, 31–34, 36, 42, 49, 50, 53, 60, 62, 68, 76, 77, 79, 88, 91, 99, 107, 109, 140, 141, 147, 148

Arbeit typischerweise eine Zahl von Flops, ein Maß für die Lesitung, die ein Programm erbringen muss. 6, 17

arithmetische Intensität Verhältnis von Arbeit zum Datenverkehr. 6

Block ein evtl. mehrdimensionaler Verbund von Threads in Software (CUDA). Jeder Block besteht aus der gleichen Anzahl an Threads. 22, 25, 27, 28, 30, 31, 86, 134

Command Queue eine Menge von Instruktionen, die der Reihe nach abgehandelt werden sollen (OpenCL). 58–60, 147

compute capability eine Versionsnummer der Chiparchitektur, die u.A. bestimmt, welche Features von CUDA auf der GPU implementiert wurden. 16, 36, 37, 46

constant Memory siehe erstes Vorkommen 23, 27, 57

Datenverkehr eine Zahl an Bytes, die während des Ausführens eines Programms insgesamt transferriert wird. 6

Gang ein evtl. mehrdimensionaler Verbund von Vectors in Software (OpenACC). Jede Gang besteht aus der gleichen Anzahl an Vectors. 134

global Memory siehe erstes Vorkommen 23, 24, 38, 57, 85, 140

Grid die Gesamtheit aller Blöcke (Nvidia). 22, 27, 28, 31

Halfwarp ein Verbund von 16 Threads in Hardware (Nvidia, CUDA Terminologie). 18

Handle ein eindeutiger Referenzwert zu einer vom Betriebssystem verwalteten Systemressource. 53, 86, 93, 97, 99, 101, 103, 108, 109, 147

Kernel eine kleine Recheneinheit, die auf die GPU ausgelagert werden soll. 20–22, 24, 25, 27, 32, 34–37, 39, 40, 50–52, 57, 58, 62, 64–66, 68, 75, 77, 78, 128, 135, 140–143, 145, 147, 148

Kontext ein Handle für das OpenCL Programm (Speicherobjekte, Programs, Command Queues, ...). 53

local Memory Speicher (cached) eines einzelnen Threads (CUDA).
ein kleiner, sehr schneller Speicher, den sich alle Workitems einer Workgroup teilen (OpenCL). Pro SM wird einer verbaut (Nvidia). 23, 57, 78

MTIU Multi Threaded Instruction Unit, ein kleiner Controller, der Instruktionen an die Warps ausgibt. 17, 22

nvcc Nvidia C(++) Compiler 46

nvlink eine High-Speed Datenschnittstelle für Nvidia Grafikkarten in Clustern. 15

NVPTX Nvidia Parallel Thread Execution, eine assemblerartige Sprache, in die nvcc Devicecode übersetzt. 46, 67, 128

page-locked Memory allozierter Speicher, der nicht vom Betriebssystem verschoben werden darf. 35

parallele Effizienz Verhältnis von Speedup zur Anzahl der eingesetzten Prozessoren. 3, 5

PCIe Peripheral Component Interconnect Express, die standard-Schnittstelle für Steckkarten. 14, 15

Peak Bandwidth maximale interne Speicherbandbreite. 6, 15

Peak Performance maximale Performance eines Prozessors. 6

Performance das Verhältnis von Flops zur benötigten Rechenzeit, ein Maß für die Effizienz von Soft- und Hardware. 7, 18, 27, 31, 38, 113

Platform eine OpenCL Imolementierung in Software. 52, 53

private Memory Speicher (cached) eines einzelnen Workitems (OpenCL). 57

shared Memory siehe erstes Vorkommen 18, 23, 25, 26, 28, 30, 37, 40

SM Streaming Multiprocessor, Eine Recheneinheit bestehend aus mehreren Warps, einer MTIU und shared memory. 17, 18, 22, 23, 31

Speedup Faktor, um den ein paralleles Programm schneller läuft, als die sequentielle Variante. 3–5

Stream eine Menge von Instruktionen, die der Reihe nach abgehandelt werden sollen (CUDA). 33–37, 39, 58, 101

Texture Memory siehe erstes Vorkommen 23

Thread die kleinste Recheneinheit einer GPU in CUDA, in Hardware auf Nvidia GPUs auch CUDA Core genannt. 17, 18, 21–23, 25, 27, 28, 30, 31, 38, 40, 42, 77, 78, 134

Vector die kleinste Recheneinheit einer GPU in OpenACC. 134

Warp Kombination zweier Halfwarps. 17, 18, 27, 30, 38, 49, 134

Wavefront ein Verbund von 32 oder 64 Workitems in Hardware (AMD). 49

Wavefront ein Verbund von 32 Vectors in Hardware (Nvidia, OpenACC Terminologie). 134

Workgroup ein evtl. mehrdimensionaler Verbund von Workitems in Software (OpenCL). Jede Workgroup besteht aus der gleichen Anzahl an Workitems. 49, 141

Workitem die kleinste Recheneinheit einer GPU in OpenCL. 57

Einführung in High-Performance-Computing

1. Einführung in High-Performance-Computing

1.1. Amdahl's Law

Gegeben sei ein sequentielles Programm. Zerlegt man dieses in Teile, die theoretisch parallelisierbar sind, und jene, die in jedem Fall sequentiell laufen müssen, so ergibt sich die Gesamtlaufzeit T als Summe

$$T = t_p + t_s \tag{1.1}$$

wobei t_p die Laufzeit des parallelisierbaren Anteils und t_s die des sequentiellen bezeichnet. Zu den unparallelisierbaren Anteilen zählen typischerweise Prozessinitialisierung oder Speicherverwaltung. Benutzt man ferner Hardware zum Ausführen des Programms, auf der insgesamt n_P Prozessoren oder Prozessorkerne dauerhaft zur Verfügung stehen, so gilt für den sogenannten maximalen Speedup S:

$$S = \frac{T}{t_s + \frac{t_p}{n_p}} \leq \frac{T}{t_s} = \frac{T}{T - t_p} \tag{1.2}$$

Dabei wird das Verhältnis $\varepsilon = \frac{S}{n_P}$ auch als parallele Effizienz bezeichnet. Als Speedup wird der Faktor bezeichnet, um den sich die Geschwindigkeit eines parallelen Programms gegenüber der sequentielle Vatiante verbessert. Abbildung 1.2 zeigt diesen Speedup in Abhängigkeit von n_p für verschiedene Verhältnisse von $\frac{t_p}{T}$. Allerdings hängt der sequentielle Anteil auch von der Parallelisierung selbst ab. Durch z.B. Prozessorkommunikation oder Synchronisation entsteht in der Gesamtlaufzeit ein zusätzlicher Summand $t_{O(n_P)}$. In diesem Beispiel steigt der Anteil linear

mit n_P. Dies ist jedoch nicht zwingend der Fall. Damit ergibt sich der maximale Speedup:

$$S = \frac{T}{t_s + t_{O(n_P)} + \frac{t_p}{n_p}} \leq \frac{T}{t_s} = \frac{T}{T - t_p} \tag{1.3}$$

Diese Kurve konvergiert nun nicht mehr gegen $\frac{T}{t_s}$, sondern erreicht ein Maximum, um danach wieder abzufallen (Abb. 1.1). In der Praxis ergibt sich ein Optimum, eine Anzahl von Prozessoren, bei der sich der maximale Speedup erzielen lässt. Eine Vergrößerung dieser Anzahl hat einen negativen Effekt, da der Geschwindigkeitsgewinn durch die weiteren Prozessoren den Aufwand der Prozessorkommunikation nicht mehr kompensiert.

1.2. Gustafson's Law

Amdahl stellte mit seinem Gesetz eine worst-case Abschätzung auf, bei der sich die Laufzeit mit der Zahl der Prozessoren linear verbessern lässt. Amdahl geht dabei von einer festen Problemgröße aus. Gustafsons Gesetz hingegen geht von einem festen Zeitfenster aus. In diesem wächst die Problemgröße mit Anzahl der Prozessoren. Mit dieser Zahl wächst linear auch die Größe der Aufgabe, die unter den Voraussetzungen gelöst werden kann.

Setzt man $P = t_p/T$ lässt sich 1.1 umschreiben:

$$1 = (1 - P) + P \tag{1.4}$$

Lässt sich der parallele Anteil unter Vernachlässigung von $t_{O(n_P)}$ gleichzeitig auf n_P Prozessoren ausführen, so gilt für den Speedup:

$$S(n_P) = (1 - P) + n_P P \tag{1.5}$$

Der parallele Anteil wächst also linear mit der Anzahl der Prozessoren. Im Gegensatz zu Amdahl wird der sequentielle Anteil mit zunehmenden n_p bedutungslos, wirkt also nicht beschränkend.

1.3. Amdahl vs. Gustafson

Nach Amdahl ist die parallele Effizienz:

$$\varepsilon_{\text{Amdahl}}(n_P) = \frac{1}{n_P} \cdot \frac{1}{(1-P) + P/n_P} = \frac{1}{n_P(1-P) + P} \tag{1.6}$$

Nach Gustafson ist die parallele Effizienz:

$$\varepsilon_{\text{Gustafson}}(n_P) = 1 - \frac{n_P - 1}{n_P} \cdot (1-P) = P + \frac{1-P}{n_P} \tag{1.7}$$

Laut Amdahl geht die parallele Effizienz im Limit einer großen Prozessorzahl immer gegen null, bei Gustafson existiert eine untere Schranke P:

$$\lim_{n_P \to \infty} \varepsilon_{\text{Amdahl}}(n_P) = 0$$
$$\lim_{n_P \to \infty} \varepsilon_{\text{Gustafson}}(n_P) = P$$

Welches Modell nun Anwendung finden sollte, hängt von der Problemstellung ab. Gustafsons Gesetz sollte auf Probleme angewandt werden, falls sich die Promblemgröße auf die Parallelisierung anpassen lässt, andernfalls Amdahl. In jedem Fall geben Amdahl und Gustafson eine Ober- und Untergrenze für den Speedup an.

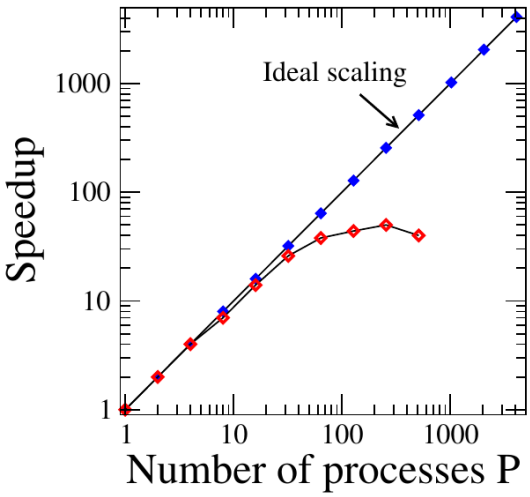

Abbildung 1.1.: Der Speedup laut Amdahl bei einer bestimmten Zahl an Prozessoren unter Berücksichtigung von Prozessorkommunikation. Ab einer bestimmten Anzahl nimmt der Speedup wieder ab. (doppelt-logarithmische Skala) [22]

1.4. Roofline Model

Sei eine Arbeit W gegeben, die ein Programm oder auch nur eine bestimmte Funktion verrichten muss. W wird üblicherweise angegeben in einer Zahl an Fließpunktoperationen (engl. floating point operations, FLOPs). Abhängig von der Problemstellung kann aber auch ein anderes Maß gewählt werden. W ist eine Eigenschaft des Programms und hängt daher nur unwesentlich von der Hardware ab.

Der Datenverkehr Q (engl. memory traffic) bezeichnet die Zahl an Bytes, die während des Ausführens des Programms insgesamt transferriert wird. Q ist im Wesentlichen eine Eigenschaft der Hardware und hängt z.B. von der Cachehierarchie ab.

$I = W/Q$ bezeichnet die arithmetische Intensität. Dabei handelt es sich um die Anzahl an Operationen pro transferriertes Byte. Die Einheit ist FLOPs/Byte. Das sogenannte naive Roofline Modell geht von nur zwei Parametern aus, der *Peak Bandwidth* β und der *Peak Performance* π. Lediglich I ist variabel. Die *Peak Performance* ist eine feste Hardwaregröße, die vom ausführenden Gerät abhängt. Diese Größe lässt sich aus den entsprechenden Datenblättern ablesen. Die *Peak Bandwidth* wird normalerweise durch *benchmarking* bestimmt (später mehr). Die er-

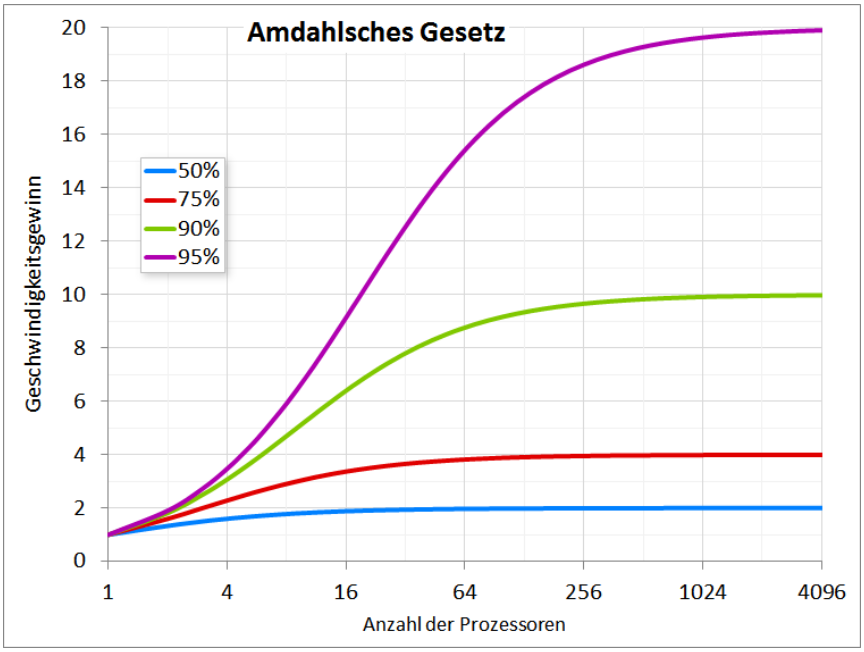

Abbildung 1.2.: Der Speedup laut Amdahl bei einer bestimmten Zahl an Prozessoren ohne Berücksichtigung von Prozesskommunikation für verschiedene Verhältnisse von t_s/T. Der Speedup konvergiert schnell gegen T/t_s. (einfach-logarithmische Skala) [2]

reichbare Performance kann dann wie folgt berechnet werden:

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases} \tag{1.8}$$

Abbildung 1.3 zeigt dieses Verhalten exemplarisch. Der Schnittpunkt der beiden Linien $I' = \pi/\beta$ wird als *ridge point* bezeichnet und charakterisiert die eingesetzte Hardware entscheidend. Er bezeichnet jenes I, das ein Programm mindestens erreichen muss, um bei gegebener Hardware diese maximal auszureizen. Gleichzeitig beschreibt er den Punkt, ab der eine Erhöhung von I keinen Effekt mehr hat. Bei Programmen, für die $I < I'$ gilt, spricht man von *memory bound*, andernfalls von *compute bound*.

Beim naiven Modell handelte es sich um eine best-case Abschätzung. In der Praxis gibt es allerdings weitere Limitierungen. Die Abbildungen 1.4, 1.5 und 1.6 zeigen folgende Fälle:

1. Kommunikation: bandwidth ceilings

2. Berechnung: in-core ceilings

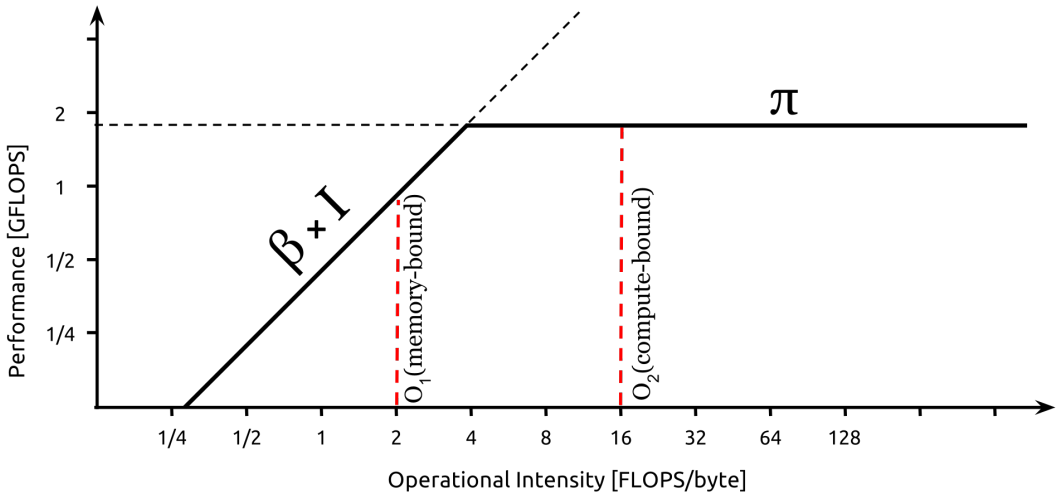

Abbildung 1.3.: Die Performance P als Funktion der arithmetischen Intensität I bei doppelt-logarithmischer Skalierung. (naives Modell) [25]

3. Lokalität: locality walls

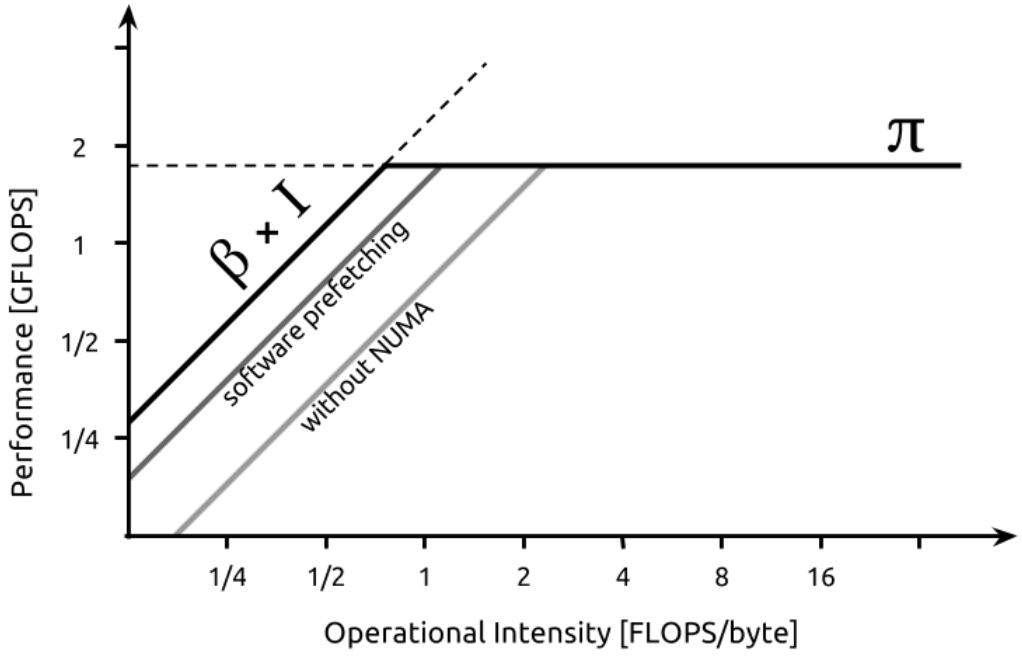

Abbildung 1.4.: Die Performance P als Funktion der arithmetischen Intensität I bei doppelt-logarithmischer Skalierung. Es existiert ein negativer Offset aufgrund von *bandwidth ceiling*. Dieses Verhalten entsteht durch fehlende Speicheroptimierung. [25]

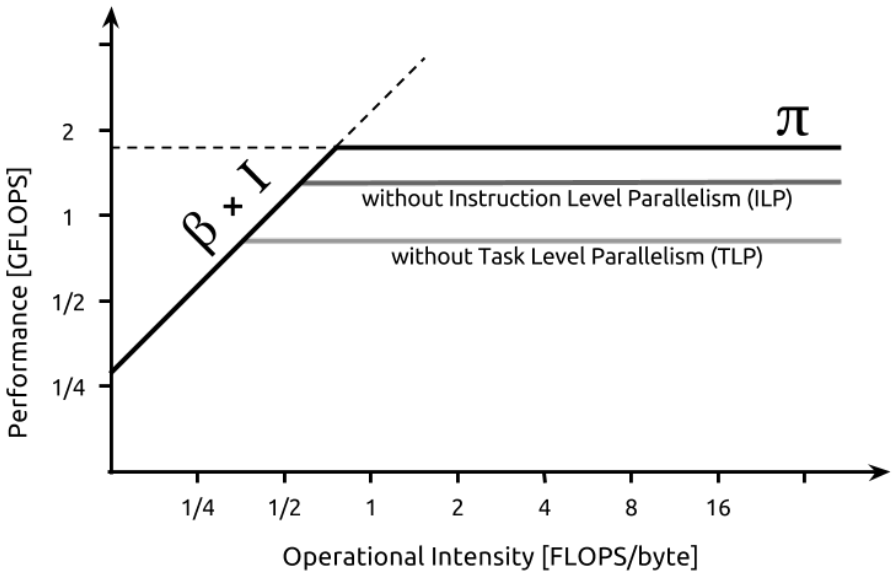

Abbildung 1.5.: Die Performance P als Funktion der arithmetischen Intensität I bei doppelt-logarithmischer Skalierung. Es existiert ein e Beschränkung der *peak performance* aufgrund von *in-core ceiling*. [25]

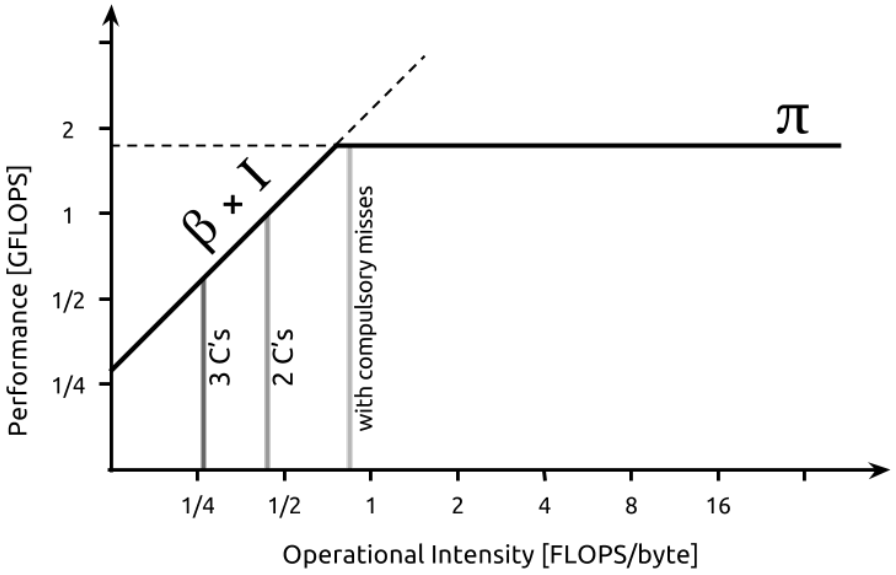

Abbildung 1.6.: Die Performance P als Funktion der arithmetischen Intensität I bei doppeltlogarithmischer Skalierung. I lässt sich aufgrund der Cachetopologie über bestimmte *locality walls* hinaus nicht steigern. rechts: *compulsory misses*, C3: zusätzlich *capacity-* und *conflict misses*, C2: zwei von drei [25]

Hardwaregrundlagen

2. Hardwaregrundlagen

2.1. Flynn's Taxonomy

Aus Wikipedia [12]:

Die Flynn'sche Taxonomie wurde 1966 von Michael J. Flynn publiziert und beschreibt eine grobe Unterteilung von Rechnerarchitekturen basierend auf der Anzahl der Befehls- (instruction streams) und Datenströme (data streams). Die Klassifikation teilt im Wesentlichen vier Bereiche ein:

- **Single Instruction, Single Data (SISD)**
  Unter SISD-Rechnern versteht man traditionelle Einkernprozessor-Rechner, die ihre Aufgaben sequentiell abarbeiten. SISD-Rechner sind z. B. Personal-Computer (PCs) oder Workstations, welche nach der Von-Neumann- oder der Harvard-Architektur aufgebaut sind. Bei ersterer wird für Befehle und Daten die gleiche Speicheranbindung verwendet, bei letzterer sind sie getrennt.

- **Single Instruction, Multiple Data (SIMD)**
  Eine Architektur von Großrechnern beziehungsweise Supercomputern. SIMD-Computer, auch bekannt als Array-Prozessoren oder Vektorprozessor, dienen der schnellen Ausführung gleichartiger Rechenoperationen auf mehrere gleichzeitig eintreffende oder zur Verfügung stehende Eingangsdatenströme und werden vorwiegend in der Verarbeitung von Bild-, Ton- und Videodaten eingesetzt.

  Dies ist sinnvoll, weil in diesen Bereichen die zu verarbeitenden Daten meist hochgradig parallelisierbar sind. So sind z. B. bei einem Videoschnitt die Operationen für die vielen einzelnen Bildpunkte identisch. Theoretisch optimal wäre hier die Ausführung durch einen einzigen, auf alle Punkte anzuwendenden Befehl.

Des Weiteren sind im Multimedia- und Kommunikationsbereich erforderliche Operationen häufig keine einfachen, einzelnen Operationen, sondern eher umfangreichere Befehlsketten. Das Einblenden eines Bildes vor einem Hintergrund ist beispielsweise ein komplexer Vorgang aus Maskenbildung mittels XOR, Vorbereitung des Hintergrundes mittels AND und NOT, sowie der Überlagerung der Teilbilder durch OR. Dieser Anforderung wird durch die Bereitstellung neuer komplexer Befehle entsprochen. So vereinigt z. B. der MMX-Befehl PANDN eine Invertierung und Und-Verknüpfung der Form x = y AND (NOT x).

Viele moderne Prozessorarchitekturen (wie PowerPC und x86) beinhalten inzwischen SIMD-Erweiterungen, das heißt spezielle zusätzliche Befehlssätze, die mit einem Befehlsaufruf gleichzeitig mehrere gleichartige Datensätze verarbeiten.

Allerdings muss man zwischen Befehlen unterscheiden, die lediglich gleichartige Rechenoperationen ausführen und anderen, die bis in den Bereich der DSP-Funktionalität hineinreichen (beispielsweise ist AltiVec in dieser Hinsicht wesentlich leistungsfähiger als 3DNow).

- **Multiple Instruction, Single Data (MISD)**
  Eine Architektur von Großrechnern bzw. Supercomputern. Die Zuordnung von Systemen zu dieser Klasse ist schwierig, sie ist deshalb umstritten. Viele sind der Meinung, dass es solche Systeme eigentlich nicht geben dürfte. Man kann aber fehlertolerante Systeme, die redundante Berechnungen ausführen, in diese Klasse einordnen. Ein Beispiel für dieses Prozessorsystem ist ein Schachcomputer.

  Eine Umsetzung ist das Makropipelining, bei dem mehrere Recheneinheiten hintereinander geschaltet sind. Eine weitere sind redundante Datenströme zur Fehlererkennung bzw. -korrektur.

- **Multiple Instruction, Multiple Data (MIMD)**
  Eine Architektur von Großrechnern bzw. Supercomputern. MIMD-Computer führen gleichzeitig verschiedene Operationen auf verschieden gearteten Eingangsdatenströmen durch, wobei die Verteilung der Aufgaben an die zur Verfügung stehenden Ressourcen, meistens durch einen oder mehrere Prozessoren des Prozessorverbandes, selbst zur Laufzeit durchgeführt wird. Jeder Prozessor hat Zugriff auf die Daten anderer Prozessoren.

  Man unterscheidet eng gekoppelte Systeme und lose gekoppelte Systeme. Eng gekoppelte

Systeme sind Mehrprozessorsysteme, während lose gekoppelte Systeme Multicomputersysteme sind.

Multiprozessorsysteme teilen sich den vorhandenen Speicher und sind somit also ein Shared-Memory-System. Diese Shared-Memory-Systeme lassen sich weiter in UMA (uniform memory access), NUMA (non-uniform memory access) und COMA (cache-only memory access) unterteilen.

Man versucht bei MIMD eine Problemstellung durch die Lösung von Teilproblemen in den Griff zu bekommen. Dabei entsteht wiederum das Problem, dass verschiedene Teilstränge des Problems miteinander synchronisiert werden müssen.

Ein Beispiel in diesem Falle wäre das UNIX-Kommando make. Hier können auch mit mehreren Prozessoren mehrere zusammengehörige Programmcodes gleichzeitig in Maschinensprache übersetzt werden.

2.2. Aufbau einer Grafikkarte

Auch wenn das Moor'sche Gesetz noch ansatzweise erfüllt ist, stagniert die Leistung von einzelnen Prozessoren seit Jahren (Abb. 2.1). Folglich versucht man viele Prozessoren zu einem großen Verbund zusammen zu schalten.

Die Idee: nutze Grafikkarten als co-Prozessoren. Eine Grafikkarte besteht aus mehreren Teilen (Abb. 2.3):

- **Bus Interface**: steuert den Datenfluss von der PCIe-Schnittstelle, Instruktionen werden an die GPU weitergeleitet, Speicherallozierungen an den Speichercontroller
- **Speichercontroller**: ein kleiner Mikroprozessor, der Speicher alloziert
- **Speicher**: beinhaltet den globalen und Texturspeicher. Die interne Speicherbandbreite liegt heutzutage bei bis zu 1TB/s (HBM2).
- **GPU**: das Herzstück der Grafikkarte. Ursprünglich war die einzige Aufgabe, der CPU 3d-Berechnungen für Texturen abzunehmen. Es existieren Grafikkarten mit zwei GPUs.

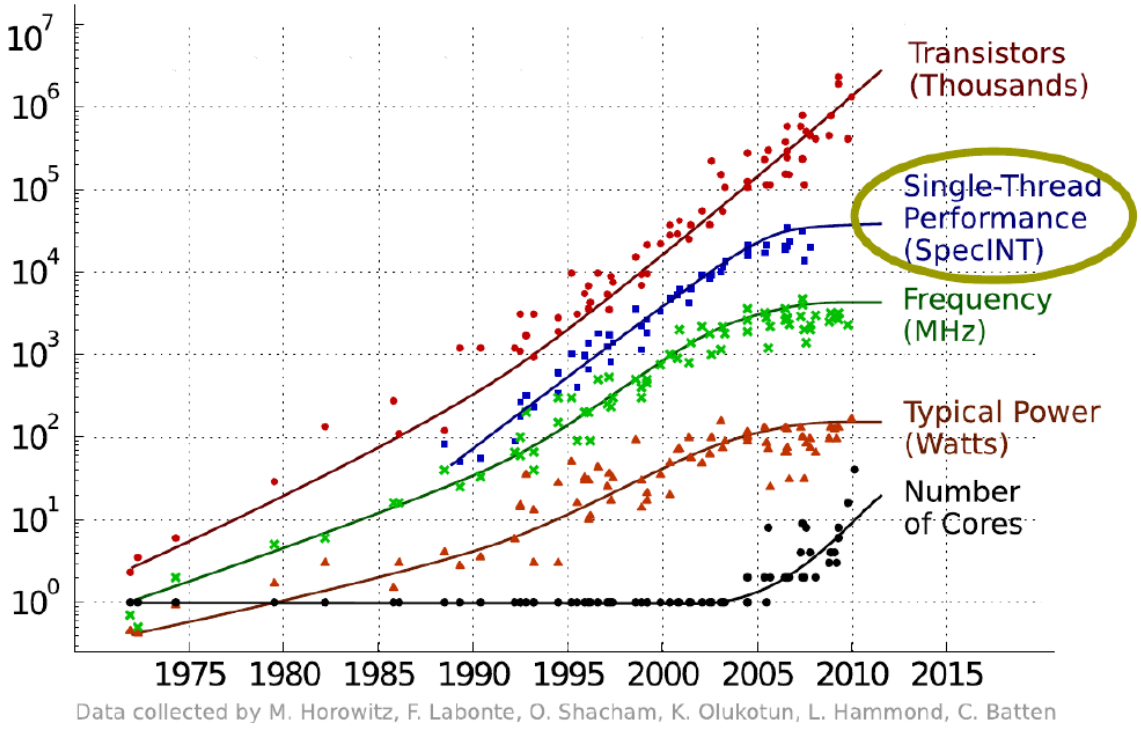

Abbildung 2.1.: Entwicklung der Hardware

- **Video-Out-Controller**: wandelt die Berechnungen der GPU in Signale für die Videoausgänge um. Ein DAC Wandler wird nur für den VGA-Ausgang benötigt und existiert auf modernen Karten nicht mehr.

Auf jeder GPU sitzt ein passiv-Kühler sowie ein seperater Lüfter. Abbildung 2.2 zeigt eine Grafikkarte verbaut in einem gewöhnlichen Desktop-Computer.

Datenverkehr zwischen CPU und GPU erfolgt meistens über PCIex16 mit einer maximalen physikalischen Bandbreite von 32GB/s (25GB/s ohne Overhead). Sehr hochwertige Karten sind über nvlink miteinander verbunden bei einer Peak Bandwidth von 300GB/s, was immer noch weit unter der internen Speicherbandbreite liegt. Kopieranweisungen zwischen CPU und GPU sollte also auf das Minimum reduziert werden.

Nvidia bietet im Wesentlichen drei Produktlinien an:

- **GeForce**: Gaming-Grafikkarten. Berechnungen für Gaming beschränken sich meist auf das Berechnen von Texturen und Ausgabe auf dem Bildschirm. Daher sind diese Karten nicht mit einem Rechenwerk für doppelte Präzision ausgestattet und verlieren einen

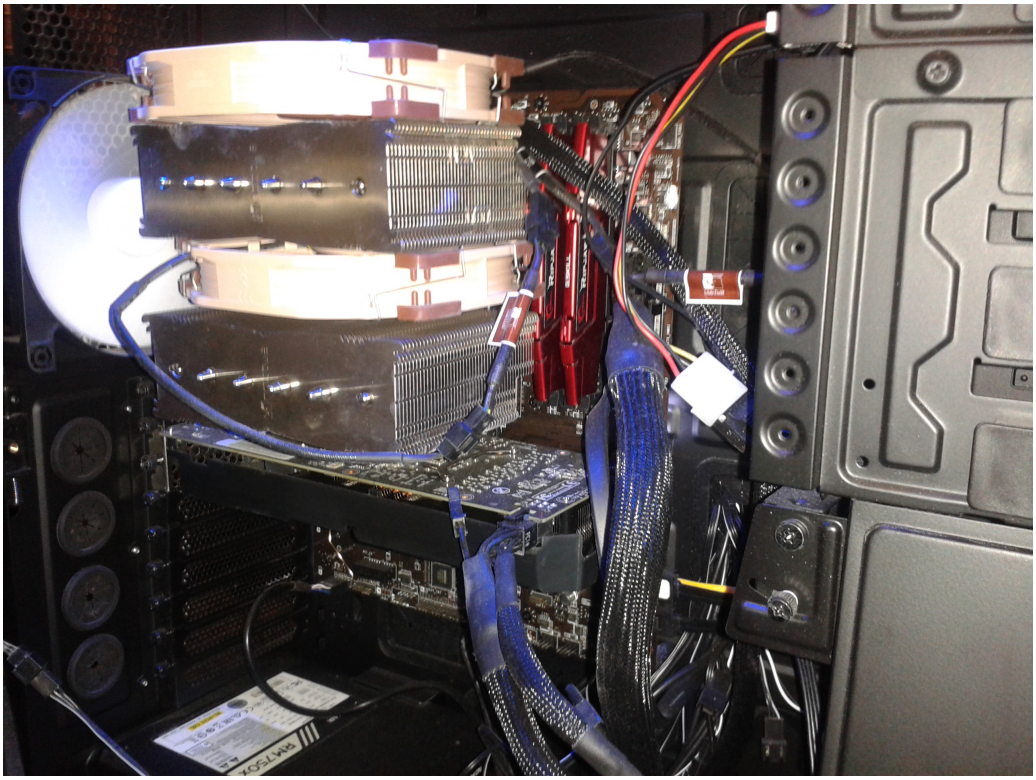

Abbildung 2.2.: Aufbau eines gewöhnlichen Desktop-Computers

erheblichen Faktor bei der Performance (≈ 32), sollte man 64Bit Arithmetik anwenden.

- **Quadro**: Ausgestattet für HPC. Diese verlieren nur den Faktor zwei bei doppelter Präzision. Verwendet werden diese für CAD (Differentialgleichungen lösen und grafisch anzeigen für Belastungsanalysen) oder grafische Simulationen.

- **Tesla** wie Quadro, aber ohne Video-Controller. Diese sind im Wesentlichen für Deep-Learning Techniken gedacht. Moderne GPUs beinhalten sogenannte Tensorkerne, die für deep-convolutional neural networks optimiert sind. Diese kommen nun vermehrt in Gaming-Karten als Unterstützung für Ray-Tracing Kerne zum Einsatz.

Karten aller drei Linien werden nach ihrer Chiparchitektur klassifiziert. Von der ältesten zur modernsten heißen diese: Tesla (nicht verwechseln!), Fermi, Maxwell, Kepler, Pascal, Volta (Weiterentwicklung: Turing)

Zudem existieren Unterkategorien, die in der sogenannten *compute capability* zum Ausdruck kommen. Diese ist die wesentlichste Kenngröße einer Nvidia Grafikkarte, da sie bestimmt, welche Features von CUDA in Hardware auf der GPU implementiert sind.

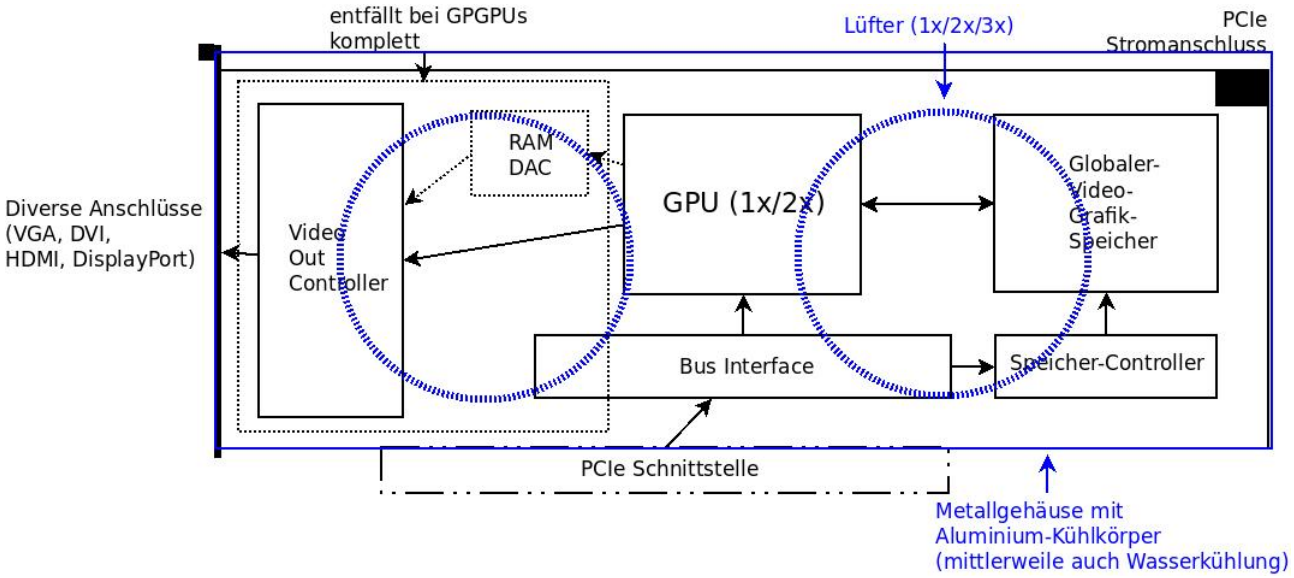

Abbildung 2.3.: Teile einer Grafikkarte

Eine GPU ist im Wesentlichen ein Verbund aus tausenden Kernen, sogenannten Threads, mit relativ geringer Leistung. Nach dem SIMD Prinzip führen alle Kerne ähliche Tätigkeiten aus. Die Idee hinter HPC auf GPUs ist, die massive Parallelität für Tätigkeiten auszunutzen, die bestimmte Eigenschaften erfüllen:

- Probleme mit extrem hoher Parallelität, z.B. sehr große unabhängige Schleifen.
- Probleme mit sehr vielen ähnlich gelagerten Berechnungen, also die selben Operationen in der gleichen Reihenfolge aber mit unterschiedlichen Daten.
- Probleme mit einer geringen Arbeit in jedem parallelen Prozess, z.B. einfache mathematische Formeln.
- Probleme mit geringem Speicherfluss.

Eine GPU wird aufgeteilt in eine bestimmte Anzahl von sogenannten Streaming Multiprocessors (SM). Üblich ist eine geringe zweistellige Anzahl (Abb. 2.4). Diese bestehen wiederum aus den folgenden Komponenten:

- Multiple Threaded Instruction Unit (MTIU): veteilt die Instruktionen auf einen Thread pro Warp

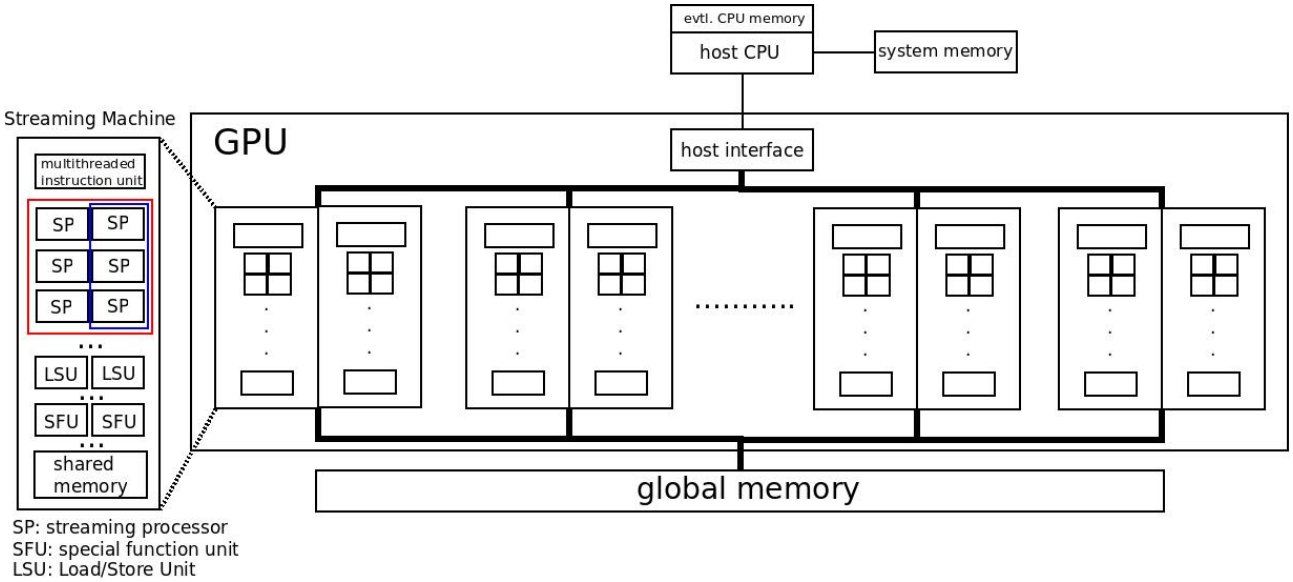

Abbildung 2.4.: Aufbau einer GPU

- Warps: ein Verbund von 32 Threads. Eine Instruktion wird kopiert und an alle weitergegeben.
- Halfwarp: Genau eine Hälfte eines Warps
- Special Function Unit (SFU): Erledigt besondere Aufgaben, z.B. shader Berechnungen (später mehr)
- Shared Memory: ein kleiner, extrem schneller Speicher, den sich alle Threads eines SMs teilen.

Es ist wichtig, die technischen Details der Hardware zu kennen, um die Software darauf anzupassen. Ein HPC Programm ist üblicherweise genau auf die Hardware zugeschnitten, auf der das Programm laufen soll. Die Generalisierung eines Problems ist nicht trivial und manchmal sogar unmöglich oder nur unter starken Performanceverlusten zu erreichen.

Einführung in Nvidia CUDA

3. Einführung in Nvidia CUDA

Seit 2007 existiert das Nvidia Produkt *Compute Unified Device Architecture* (CUDA). Dabei handelt es sich um eine Programmiersprache, eine Compilerarchitektur, eine Runtime-Library oder kurz gesagt um eine Platform. CUDA lässt sich mit dem *Nvidia CUDA Toolkit* leicht auf Linux, MacOS oder Windows installieren und bringt dabei mehrere HPC Bibliotheken mit. Auf einige soll später noch eingegangen werden. Die wichtigsten Vertreter sind:

- **cuBLAS**: Die CUDA Implementierung der *Basic Linear Algebra Subprograms*.
- **curand**: Ein Zufallszahlen-Generator (RNG) und verschiedene Verteilungen.
- **cuSOLVER**: Eine Implementierung der beliebtesten Matrixzerlegungen.
- **THRUST**: Die CUDA Implementierung der C++ Standard Template Library (STL).

Andere prominente Vertreter sind cuDNN, TensorRT und offizielle Partnerprojekte wie MAGMA, GROMACS oder Tensorflow.

3.1. C Runtime

Ein Hauptprogramm auf der CPU wird üblicherweise in C oder C++ geschrieben. Wahlweise lässt sich auch ein API in einer anderen Programmiersprache benutzen, z.B. pyCUDA in Python. Dieses Programm läuft wie gewohnt auf der CPU und wird fortan als Hostcode bezeichnet. Eine sehr rechenintensive Stelle im Hostcode soll nun auf die GPU ausgelagert werden. Im gleichen Quellcode wird nun in der Programmiersprache CUDA ein sogenanntes Kernel definiert.

Kernels

Kernel sind sehr kleine Recheneinheiten, die parallel auf der GPU ausgeführt werden sollen. Der folgende Code zeigt ein Kernel zur Addition zweier Vektoren.

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

...

int main()
{
  ...
  VecAdd<<<1, N>>>(A, B, C, N) ;
  ...
}
```

Codebeispiel 3.1: Vektoraddition Kernel

Kernel werden immer mit dem Keyword __global__ deklariert und der Rückgabetyp ist immer void. Kernelaufrufe erfolgen asynchron, d.h. die CPU schickt den Auftrag lediglich ab und arbeitet dann weiter den Hostcode ab. Mehrere Kernelaufrufe hintereinander werden auf der GPU serialisiert.

Die spitzen Klammern geben an, mit welchen Threads der Code ausgeführt werden soll. Im obigen Fall wird das Kernel mit N Threads gestartet. Ausnahmslos jeder dieser Threads erhält das Kernel und führt den Code aus: Jeder Thread rechnet sich seinen Index aus (entspricht dem Prozessor-rank). Jeder Thread nimmt einen Wert von A und B, rechnet die Summe aus und schreibt genau einen Wert von C. Ein Kernel kann man sich also so vorstellen, als würde man eine Schleife ausrollen und jedes Element einzeln und parallel berechnen. Kernel können nicht auf Hostcode zugreifen. Sollte N größer sein als die Länge der Vektoren, so müssen die überzähligen Threads warten. Ein Weglassen der if-Abfrage würde jedoch nicht zwingend zu einem Fehler führen. Um falsche Speicherzugriffe zu vermeiden, kann das Tool cuda−memcheck verwendet werden. (später mehr)

Zur besseren Steuerung der Threads teilt man diese in Blöcke ein[1]. Die Gesamtheit an Blöcken bezeichnet man als Grid.

```
int main()
{
  int threadsPerBlock = 32;
  int blocksPerGrid = N / threadsPerBlock;

  ...

  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(A, B, C, N);

  ...
}
```

Codebeispiel 3.2: Kernelaufruf

In diesem Beispiel übergibt man im ersten Element zwischen den spitzen Klammern die Zahl an Blöcken und im zweiten die Zahl an Threads pro Block. Beide Zahlen multipliziert ergeben also die Zahl an Threads insgesamt. Mittels blockIdx.x*blockDim.x + threadIdx.x berechnet man den globalen Index eines Threads. Die MTIUs verteilen die Blöcke dann automatisch auf die SM. Sollten mehr Threads involviert werden als tatsächlich zur Verfügung stehen, so wird zuerst eine passende Anzahl verteilt und dann für die restlichen Blöcke wiederholt. Der Index zählt dabei korrekt weiter, da er von der ID des Blocks abhängt.

Falls das Kernel eine Funktion aufrufen soll, so muss diese extra für das Device geschrieben werden. Dazu deklariert man diese Funktionen mit dem Keyword __device__. Analog dazu werden gewöhnliche Funktionen auf der CPU mit __host__ deklariert. Wird kein Keyword geschrieben, so handelt es sich um eine Host Funktion. Wird eine Funktion mit beidem deklariert, so wird die Funktion für Device und Host kompiliert. Device Funktionen können nicht vom Host gerufen werden und umgekehrt. Eine Device Funktion darf einen return Wert haben.

[1] Meistens existieren technische Obergrenzen für die Zahl an Threads pro Block.

C++ Kompatibilität

Die Programmiersprache CUDA orientiert sich an C11. Es können alle Features dieses Standards benutzt werden. C++ Code als Device Code zu schreiben ist allerdings wesentlich schwieriger und wurde daher nur teilweise implementiert. Eine Liste der C++ Features enthält der CUDA Programming Guide in Anhang F. [7]

3.2. Speichermodell

Eine GPU kennt mindestens fünf Arten von Speicher:

- Global Memory: Daten, die vom CPU Speicher kopiert werden, werden in diesem gespeichert. Dieser Speicher ist ausnahmslos allen Threads zugänglich.
- Constant Memory: Ein ebenfalls globaler Speicher. Konstante Variablen werden so read-only abgespeichert und verhalten sich ähnlich den konstanten Variablen in C (Vermeidung von Bankkonflikten).
- Local Memory: Jeder Thread verfügt über einen sehr kleinen Speicher für Hilfsvariablen, die nur diesem zugänglich sind. Definiert man eine Variable im Devicecode, so wandert sie in den lokalen Speicher (z.B. der Index `int i`). Dieser Speicher ist ähnlich langsam wie global Memory, wird aber in einen Cache geladen.
- Shared Memory: Ein kleiner, extrem schneller Speicher, den sich jeweils die Threads einer SM teilen. Pro SM ist ein Element in der GPU integriert.
- Texture Memory: Ein interpolierender, globaler Speicher, der für Texturen, also multidimensionale Speicherstrukturen optimiert wurde.

Um die Vektoraddition von oben durchzuführen, muss die CPU zunächst die Vektoren in den Global Memory kopieren. Folgendes Beispiel illustriert dies:

```
    uint N = 256;
2   uint size = *Nsizeof(float);
    float *h_A = (float *)malloc(size);
4   float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);
6
    float *d_A;
8   cudaMalloc(&d_A, size);
    float *d_B;
10  cudaMalloc(&d_B, size);
    float *d_C;
12  cudaMalloc(&d_C, size);


14
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
16  cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

18  int threadsPerBlock = 32;
    int blocksPerGrid = N / threadsPerBlock ;
20  VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
22
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
24  free(h_A); free(h_B); free(h_C);
```

Codebeispiel 3.3: Vektoraddition Host

Der Befehl cudaMalloc erstellt einen Buffer der entsprechenden Größe im Global Memory. Der Befehl CudaMemcpy kopiert dann den Speicher des Hostvektors in den Devicevektor auf der Grafikkarte. Dabei gibt size die Anzahl an Bytes an, die kopiert werden soll. Für eine einzelne Variable (N) ist dies nicht notwendig. Nachdem das Kernel abgeschickt wurde, wird der Ergebnisvektor auf den Host zurückkopiert. Auf die Pointervariablen lässt sich mittels Pointeraddition ein Offset angeben, die Größe muss aber entsprechend angepasst werden. Beispielhaft kopiert (d_A+3, h_A+5, 20, ...) exakt 20 Bytes beginnend von h_A[5] nach d_A[3]. Hinter cudaMemcpy verbirgt sich ein impliziter Kernelaufruf, der synchron abläuft, d.h. die CPU muss warten bis das Kernel ausgeführt wurde. Es ist zu beachten, dass Kernelaufrufe normalerweise asynchron laufen, also die CPU nicht blockieren (non-blocking). Speicher auf dem Device wird mit cudaFree

freigegeben.

Shared Memory wird mit dem Keyword __shared__ deklariert. Der erste Thread, der in einem Kernel auf eine solche Instruktion trifft, legt eine Variable im shared Memory der entsprechenden Größe an. Diese Variable teilen sich dann alle Threads eines Blocks. Existieren insgesamt n Blöcke, so existieren auch n verschiedene Variablen. In der statischen Variante muss die Größe des Arrays zur Compile-Zeit bekannt sein.

```
__global__ void test()
{
    __shared__ int s[64];
    ...
}

```

Codebeispiel 3.4: shared Memory statisch

Im Gegensatz zur statischen Variante lässt sich shared Memory auch dynamisch allozieren:

```
__global__ void test()
{
    extern __shared__ int s[];
    ...
}

int main()
{
  ...
  int shared_memory_size = ...
  test<<<1, threadsPerBlock, shared_memory_size>>>();
  ...
}

```

Codebeispiel 3.5: shared Memory dynamisch

Dem Kernel muss die Größe des shared Memory (pro Block) mitgegeben werden. In beiden Fällen wird also die Allozierung des Speichers von der CPU übernommen. Dynamische Speicherverwaltung ist mittlerweile auch auf dem Device möglich. Darauf wird jedoch erst später eingegangen (siehe Abschnitt **??**).

Sollen mehrere Arrays im shared Memory abgelegt werden, so definiert man sich lediglich

Pointer auf ein großes Gesamtarray. Der Beginn des jeweils neuen Arrays sollte sinnvollerweise dem Ende des vorangegangenen entsprechen.

```
extern __shared__ int s[];
int *integerData = s;
float *floatData = (float*)&integerData[nI];
char *charData = (char*)&floatData[nF];

...

test<<<1, threadsPerBlock,
  nI*sizeof(int) + nF*sizeof(float) + nC*sizeof(char)>>>(...);
```

Codebeispiel 3.6: shared Memory verteilt

Im folgenden Beispiel wird die Vektoraddition umgeschrieben, um sich die Vektoren A und B in den shared Memory zu laden:

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int tid = threadIdx.x;

    __shared__ float As[blockDim.x];
    __shared__ float Bs[blockDim.x];

    if (i < N)
    {
        As[tid] = A[i];
        Bs[tid] = B[i];

        ...

        C[i] = As[tid] + B[tid];
      }
}
```

Codebeispiel 3.7: Vektoraddition shared Memory

Da ohnehin einmal A[i] und B[i] aus dem Speicher geladen werden müssen, ist es hier unsinnig, den Speicher überflüssigerweise zu kopieren. Allerdings könnten ja vor der Addition As und Bs noch mehrmals gebraucht werden. Die Aufgabe des Programmierers ist es nun herauszufinden, bei welcher Datengröße abhängig vom Algorithmus der Performancegewinn durch die höhere Bandbreite den Verlust durch das Kopieren übersteigt.

Constant Memory wird mit dem Keyword __constant__ als globale Variable deklariert. Mit dem Befehl cudaMemcpyToSymbol wird diese dann in den Speicher der GPU kopiert und steht dort dann auch global zur Verfügung, muss also beim Kernelaufruf nicht explizit angegeben werden:

```
__constant__ float d_A[12800];
__global__ void VecAdd(float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = d_A[i] + B[i];
}

...

cudaMemcpyToSymbol(d_A, h_A, size);
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_B, d_C, N);
```

Codebeispiel 3.8: Vektoraddition constant Memory

Der constant Memory ist ebenfalls ein globaler Speicher und steht jedem Thread zur Verfügung. Allerdings müssen Aufrufe der selben Stelle im Speicher zur selben Zeit innerhalb eines Warps (Bankkonflikt) nicht serialisiert werden. Daher eignet sich dieser Speicher besonders für konstante Skalare, z.B. Naturkonstanten.

3.3. Mehrdimensionale Blöcke

Bisher wurden Kernel immer nur mit eindimensionalen Blöcken und Grids ausgeführt. Allerdings verbirgt sich dahinter ein Typecast. Zum Beispiel wird <<<1,N>>> zu <<<dim3(1,1,1), dim3(N,1,1)>>>. dim3 bezeichnet dabei eine eingebaute Datenstruktur, die aus drei vorzeichenlo-

sen Ganzzahlen besteht. In diesem Fall gibt sie die Anzahl der Blöcke im Grid bzw. der Threads pro Block in die Raumrichtungen (x,y,z) an. Das Grid und die Blöcke sind also keine Ketten sondern Quader von Objekten. Die Indizes der Blöcke und Threads lassen sich dann Spalten-, Zeilen- und Ebenenweise abfragen:

```
__global__ test()
{
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  ...
}

  ...

  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid(GRID_SIZE, GRID_SIZE;
  test<<<dimGrid, dimBlock>>>();
```

Codebeispiel 3.9: Multidimensionale Blöcke

In diesem Fall werden also Blöcke der Größe BLOCK_SIZE×BLOCK_SIZE in einem Grid mit GRID_SIZE×GRID_SIZE Blöcken angeordnet. Bei modernen GPUs ist die Obergrenze für die Dimension der Blöcke (1024,1024,64).

Das folgende Beispiel behandelt die Matrixmultiplikation unter Zuhilfenahme von shared Memory und Multidimensionalen Blöcken:

```
#define BLOCK_SIZE 16
typedef struct
{
  uint width;
  uint height;
  float* elements;
  uint stride;
} Matrix;

__device__
float GetElement(const Matrix A, uint row, uint col)
{
  return A.elements[row * A.stride + col];
}
```

```
16      __device__
        void SetElement(Matrix A, uint row, uint col, float value)
18      {
          A.elements[row * A.stride + col] = value;
20      }

22      __device__
        Matrix GetSubMatrix(Matrix A, uint row, uint col)
24      {
          Matrix Asub;
26        Asub.width    = BLOCK_SIZE;
          Asub.height   = BLOCK_SIZE;
28        Asub.stride   = A.stride;
          Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
30          + BLOCK_SIZE * col];
          return Asub;
32      }

34      __global__
        void MatMulKernelShared(const Matrix A, const Matrix B, Matrix C)
36      {
            uint blockRow = blockIdx.y;
38          uint blockCol = blockIdx.x;

40          Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

42          float Cvalue = 0;

44          uint row = threadIdx.y;
            uint col = threadIdx.x;
46
            for (uint m = 0; m < (A.width / BLOCK_SIZE); ++m)
48          {
            Matrix Asub = GetSubMatrix(A, blockRow, m);
50          Matrix Bsub = GetSubMatrix(B, m, blockCol);

52          __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
            __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
54          As[row][col] = GetElement(Asub, row, col);
            Bs[row][col] = GetElement(Bsub, row, col);
56
            __syncthreads();
```

```
58
            for (uint e = 0; e < BLOCK_SIZE; ++e)
60              Cvalue += As[row][e] * Bs[e][col];

62          __syncthreads();
        }
64
        SetElement(Csub, row, col, Cvalue);
66  }
```

Codebeispiel 3.10: Matrixmultiplikation

Abbildung 3.1 zeigt klar, dass shared Memory die bessere Alternative zu einer naiven Implementierung ist. Die Funktion __syncthreads() wird in Abschnitt 3.5 besprochen.

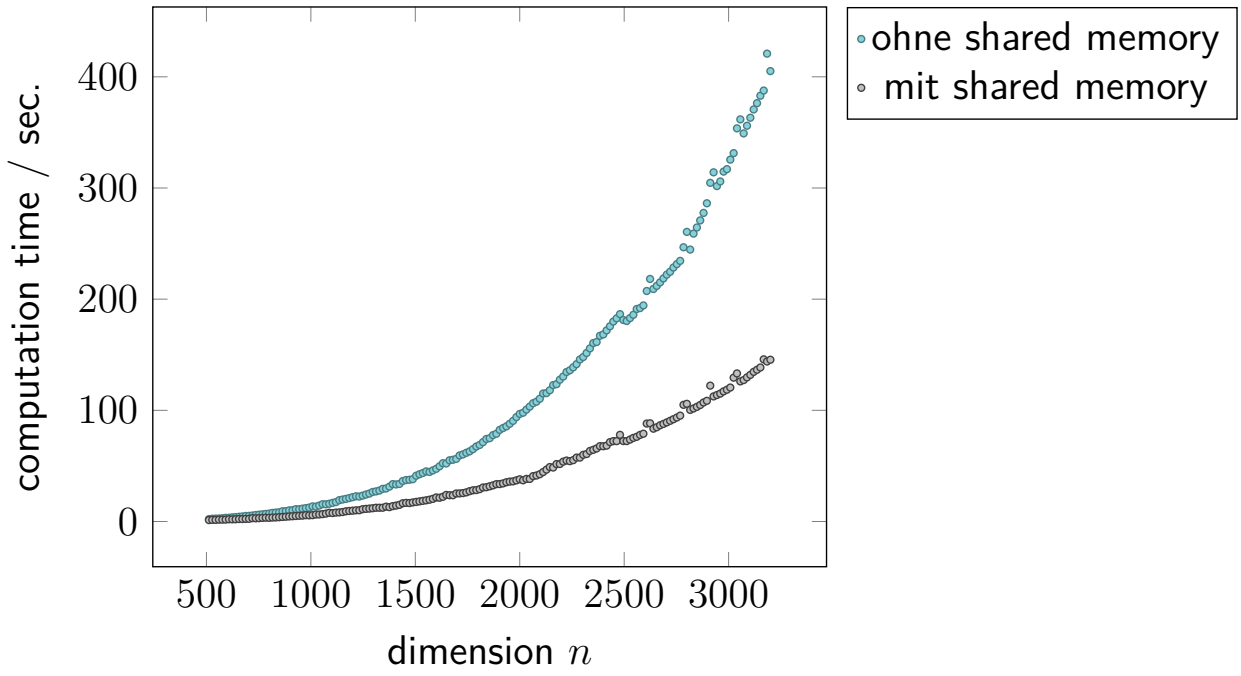

Abbildung 3.1.: Laufzeitvergleich von Matrixmultiplikationen der Größ e $n \times n$ mit und ohne shared Memory. Zum Einsatz kam eine *Nvidia GTX 1060*.

Dieser Algorithmus ist nicht ohne Weiteres anwendbar, falls sich die Matrizen nicht bequem auf gleich große Blöcke verteilen lassen.

Aus den genannten Beispielen ergeben sich Merkregeln für die Wahl der Threadzahl pro Block:

- Die Anzahl sollte ein Vielfaches von 32 sein, um unregelmäßiges Arbeiten der Warps zu

verhindern.

- Die Anzahl sollte ein Vielfaches der Anzahl von Threads pro SM sein, um unregelmäßiges Mapping von physischem Speicher in den selben Adressraum zu verhindern.
- Die Anzahl sollte so gewählt sein, dass sich die gesamte Problemgröße, also ein oder mehrere Grids, exakt auf gleich große Blöcke verteilen lässt.
- Die Gesamtzahl der Threads im Grid sollte ein Vielfaches der insgesamt in Hardware zur Verfügung stehenden Threads sein, um die GPU voll auszulasten.

Da in der Praxis diese Punkte kaum alle einzuhalten sind, ist ein Blackbox-Verfahren für die GPU schwer oder gar nicht zu programmieren. Lässt sich die Zahl der Werte N nicht bequem auf Blöcke und Threads verteilen, so muss N / threadsPerBlock auf die nächst größere Zahl aufgerundet werden. Dazu berechnet man mit einer Integerdivision blocksPerGrid = (N + threadsPerBlock − 1) / threadsPerBlock. Wenn möglich sollte man bereits bei der Erzeugung von Messdaten in einem Experiment oder in einer Simulation auf vernünftige Problemgrößen achten, andernfalls muss man mit Performanceeinbußen rechnen.

3.4. Error Handling

Beinahe jeder API-Call liefert einen speziellen CUDA-Datentyp namens cudaError_t. Folgende Funktion fragt diesen Wert ab und gibt einen Fehlerstring zurück:

```
#define CUDA_ERROR_CHECK

#define CudaCheckError()  __cudaCheckError(__FILE__, __LINE__)

inline void __cudaCheckError(const char *file, const int line)
{
#ifdef CUDA_ERROR_CHECK
  cudaError err = cudaGetLastError();
  if(cudaSuccess != err)
  {
    fprintf(stderr, "cudaCheckError() failed at %s:%i : %s\n",
    file, line, cudaGetErrorString(err));
    exit(-1);
  }
```

```
16          err = cudaDeviceSynchronize();
            if(cudaSuccess != err)
18          {
                fprintf(stderr, "cudaCheckError() with sync failed at
20                %s:%i : %s\n",
            file, line, cudaGetErrorString(err));
22          exit(-1);
          }
24      #endif

26        return;
        }
28
```

Codebeispiel 3.11: Error Handling

Diese Funktion wurde nicht im API implementiert, da das Errorhandling vom Design des gesamten Programms abhängt und daher dem Programmierer überlassen werden sollte. Funktionen dieser Art kursieren in mehreren Varianten in Foren und verschiedenen Handbüchern. Mittels cudaGetLastError wird der letzte Fehler untersucht. Da Kernel asynchron laufen, kann es nötig sein, mittels cudaDeviceSynchronize zu synchronisieren, d.h. alle Kernel müssen ausgeführt worden sein, bevor die CPU weiter Code ausführen darf (siehe Abschnitt 3.5). Falls dies nicht nötig ist, sollte die Zeile kommentiert werden. Um Rechenzeit zu sparen, kann durch das CUDA_ERROR_CHECK Makro diese Funktion zur Compile-Zeit entfernt werden. CUDA Librarys (z.B. cuDNN oder cuRAND) definieren oft neue Datentypen für Fehler. Diese Funktion kann also in mehreren Variationen nötig sein.

3.5. Stream Processing

Events

Um die Laufzeit zu messen, eignen sich CUDA-Events:

```
      cudaEvent_t start, stop;
2     cudaEventCreate(&start);
      cudaEventCreate(&stop);
```

```
4
        cudaEventRecord(start, 0);
6       //zu messender Code//
        cudaEventRecord(stop, 0);
8
        cudaEventSynchronize(stop);
10      float milliseconds = 0;
        cudaEventElapsedTime(&milliseconds, start, stop);
12
        cudaEventDestroy(start);
14      cudaEventDestroy(stop);
```

Codebeispiel 3.12: Events

Mit cudaEventRecord werden zwei Events aufgezeichnet. Die Null am Ende teilt die Events dem Stream mit Nummer 0 zu (siehe 3.5). Wird diese Angabe unterdrückt, so werden die Aufrufe automatisch diesem Stream zugeordnet (default-Stream). Nach dem Aufruf cudaEventSynchronize, also sobald das Event stop stattgefunden hat, kann mittels cudaEventElapsedTime die vergangene Zeit zwischen start und stop in Millisekunden gemessen werden.

Zudem existiert cudaError_t cudaEventQuery(cudaEvent_t event). Diese Funktion liefert dann einen Erfolg, falls das betreffende Event stattgefunden hat.

Außerdem lassen sich mittels cudaError_t cudaEventCreateWithFlags(cudaEvent_t* event, unsigned int flags) Events mit Flags definieren, die sich in der Dokumentation der CUDA Runtime API nachlesen lassen. [8]

Device Auswahl

Mittels cudaGetDeviceCount lässt sich ermitteln, über wie viele CUDA-fähige Grafikkarten die Platine, auf der die CPU sitzt, verfügt. Für jedes Gerät einzeln können dann die Eigenschaften des Geräts abgefragt werden.

Die Geräte lassen sich mit cudaSetDevice manuell umschalten. Speicher der einen GPU lässt sich nur dann von der anderen abrufen, wenn der Peer-to-Peer Access mit cudaDeviceEnablePeerAccess aktiviert und der Speicher explizit zwischen den GPUs kopiert wird. Die Null am Ende teilt

die Events dem Stream mit der Nummer 0 zu (siehe 3.5). Wird diese Angabe unterdrückt, so werden die Aufrufe automatisch diesem Stream zugeordnet (default-Stream).

```
    int deviceCount;
2   cudaGetDeviceCount(&deviceCount);

4   cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);
6   printf("Device %d has compute capability %d.%d.\n",
        device, deviceProp.major, deviceProp.minor);
8
    cudaSetDevice(0);
10  float* p0;
    cudaMalloc(&p0, size);
12  test<<<...>>>(p0);

14  cudaSetDevice(1);
    float* p1;
16  cudaMalloc(&p1, size);
    test<<<...>>>(p1);
18


20

    cudaDeviceEnablePeerAccess(0, 0);
22  test<<<...>>>(p0);

24  cudaMemcpyPeer(p1, 1, p0, 0, size);
```

Codebeispiel 3.13: Device Peer-to-Peer Access

Streams

Eine Menge von Instruktionen, die der Reihe nach abgearbeitet werden sollen, heißt Stream. Jene Streams lassen sich beliebig erstellen und zerstören (cudaStreamCreate, cudaStreamDestroy). In folgendem Beispiel wird dynamisch eine Zahl von Streams erstellt. Dann wird ein entsprechender Teil eines Gesamtarrays kopiert und ein Kernel ausgeführt. Damit dies in Reihe geschieht, werden sowohl API-calls als auch Kernel einem Stream zugeordnet.

```
uint stream_num = ...;
uint size = ...;
float* hostPtr;
cudaMallocHost(&hostPtr, stream_num * size);
//inputDevPtr allozieren und kopieren//
cudaStream_t stream[stream_num];
for (int i = 0; i < stream_num; ++i)
{
  cudaStreamCreate(&stream[i]);

  cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
    size, cudaMemcpyHostToDevice, stream[i]);

  MyKernel <<<100, 512, 0, stream[i]>>>
    (outputDevPtr + i * size, inputDevPtr + i * size, size);

  cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
    size, cudaMemcpyDeviceToHost, stream[i]);

  cudaStreamDestroy(stream[i]);
}
```

Codebeispiel 3.14: Streams

Die Idee dahinter ist, die CPU möglichst wenig zu blockieren (Asynchronizität, non-blocking) und möglichst viele Kernel gleichzeitig auszuführen (Concurrency, siehe 3.5). Da ein normaler Kopierbefehl automatisch synchronisiert, muss die Kopie asynchron erfolgen (cudaMemcpyAsync). Mittels cudaMallocHost wird dafür sog. page-locked Memory alloziert. Dieser Speicher darf vom Betriebssystem nicht verschoben werden und kann daher immer unter der selben Speicheradresse gefunden werden. Kernel werden automatisch non-blocking ausgeführt.

Wird kein Stream angegeben, so wird der default-Stream verwendet.

Um die CPU auf einen bestimmten Stream warten zu lassen, kann cudaStreamSynchronize verwendet werden. cudaStreamQuery liefert genau dann einen Erfolg, wenn der Stream erfolgreich beendet worden.

cudaStreamWaitEvent beginnt mit der Ausführung eines Streams erst, sobald ein bestimmtes

Event aufgezeichnet wurde. Dieses Event kann ebenfalls einem Stream zugeordnet werden (cudaEventRecord(start, stream[i])).

So lassen sich z.B. Abfragen erstellen,

```
if ((cudaStreamQuery(stream[i]) == cudaSuccess) && (cudaEventQuery(stop) == cudaSuccess))...
```

mit denen man in Kombination mit der Device-Auswahl ganze Bäume von Kernelaufrufen erstellen kann. Auf Clustern kann man sich dies zu Nutze machen, um exakt den zeitlichen Ablauf von Kernels zu steuern. Anweisungen, die sich im selben Stream befinden, werden in Reihe abgearbeitet. Um dieses Verhalten zu verhindern, kann cudaStreamCreateWithFlags(&stream , cudaStreamNonBlocking) verwendet werden.

Wichtig:

Funktionen, die dem default-Stream zugeordnet werden, haben bezüglich Synchronizität auf der GPU ein anderes Verhalten. Es gilt, dass alle API Aufrufe und Kernel in ihrer Reihenfolge auf dem Device abgehandelt werden. Für eigene Streams ist dies nicht zwingend der Fall (siehe Abschnitt 3.5).

Concurrency

Moderne GPUs bieten die Möglichkeit gleichzeitiger Kernelaufrufe an. Werden Kopierbefehle und Kernelaufrufe verschiedenen Streams zugeordnet, so können diese gleichzeitig auf dem Device ausgeführt werden. Dieses Verhalten kann jedoch nicht garantiert werden und kann daher nicht für Probleme verwendet werden, die zwingend Concurrency erfordern. Zudem gibt es für die Anzahl an gleichzeitig ausführbaren Kernel eine Obergrenze, die bei jeder Compute capability anders ist.

Diese Möglichkeit bietet einige Vorteile:

- Probleme, die laut Amdahl kleiner Parllelität erfordern, können so in kleinerem Umfang ausgeführt werden, ohne Hardware Kapazitäten zu vergeuden, da sich mehrere von diesen Problemen gleichzeitig ausführen lassen.

- PCIe ist *full-duplex*. Es kann gleichzeitig gelesen und geschrieben werden. Dazu müssen

sich aber beide Kopiervorgänge in verschiedenen Streams befinden. In diesem Fall erhält man fast die doppelte Bandbreite.

- Wenn mehrere Kernel ausgeführ werden sollen, ihre Reihenfolge aber egal ist, kann so der GPU überlassen werden, welche Operationen wann ausgeführt werden.

Synchronisation

Explizite Synchronisierungsfunktionen:

```
__host__ cudaError_t cudaDeviceSynchronize(void)
__host__ cudaError_t cudaStreamSynchronize(cudaStream_t)
__host__ cudaError_t cudaEventSynchronize(cudaEvent_t)
__device__ void __syncthreads()
__device__ void __syncwarp()

```

Codebeispiel 3.15: Explizite Synchronisierung

Erstere bildet für die CPU eine Barriere, bis sämtliche Kernels abgearbeitet wurden.

Implizite Synchronisierung:

- page-locked Host Memory Allozierung
- Device Memory Allozierung
- Setzen von Device Memory
- Kopie zwischen zwei Adressen innerhalb desselben Device Memory
- ein beliebiges CUDA Kommando den 0-Stream betreffend,
- ein Wechsel zwischen L1/shared Memory Konfigurationen beschrieben in compute capability 3.x und compute capability 7.x

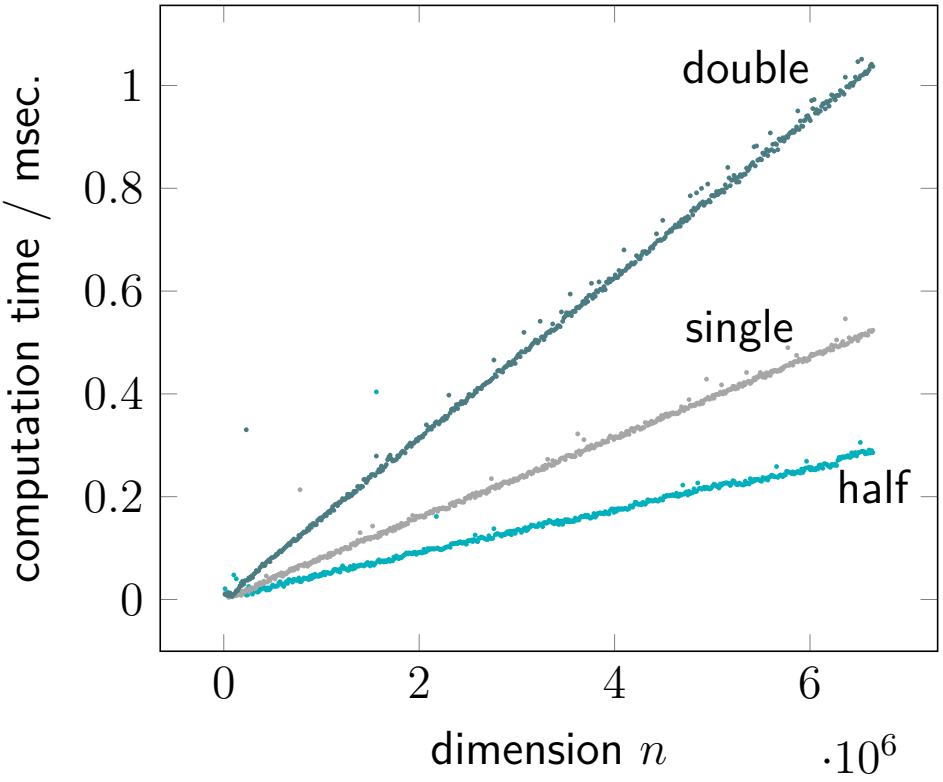

Abbildung 3.2.: Laufzeitvergleich von Axpy-Operation ($\vec{y} \to a \cdot \vec{x} + \vec{y}$) der Größe n mit half, single- und double-Präzision. Zum Einsatz kam eine *Nvidia GTX 1060*.

Divergenz

Synchronisation ist nich nur aufgrund von Abhängigkeiten in Algorithmen notwendig. Threads innerhalb eines Warps müssen zur selben Zeit die selben Instruktionen erhalten. Ist dies im Programm nicht der Fall, z.B. durch if-else Verzweigungen, müssen diese Anweisungen serialisiert werden. Führen innerhalb eines jeden Warps die Threads paarweise unterschiedliche Instruktionen aus, so verliert man den Faktor 32 in der Performance. Diese sogenannte Divergenz kann nach jeder if-Anweisung oder abgebrochener Schleife auftreten und sollte so schnell wie möglich durch Synchronisierung aufgelöst werden.

3.6. große Datenmengen

Sollten die Daten, die für eine bestimmte Rechnung nötig sind, zu groß für den Global Memory der GPU sein, so muss das Problem in kleinere Einheiten aufgeteilt werden. Bei der Vektorad-

dition ist dies kein Problem, da der Input in kleinere Teile geteilt und einzeln addiert werden kann.

Andere Probleme erfordern aber die kompletten Daten, zum Beispiel die Matrixmultiplikation. Bei komplexeren Problemen wird es zur Hauptaufgabe des Programmierers, den Algorithmus so anzupassen, dass dieser aufgeteilt werden kann und die verschiedenen Kernel miteinenader wechselwirken können. Ein einfaches Beispiel dafür wäre die Summenreduktion, bei der von Teilarrays die Summe gebildet wird und diese Teilsummen dann wieder parallel aufsummiert werden (siehe Abschnitt 3.8).

Da nun für die Daten nur ein zu kleines Array bereitsteht, in das mmer wieder geschrieben werden muss, müssen die Anweisungen wieder in Reihe geschehen. Allerdings können sich die Kopie zurück auf den Host und die folgende Kopie auf das Device immer noch überlappen. Folgendes Beispiel modifiziert jenes aus Abschnitt 3.5 so, dass alle Operationen auf Teilarrays in Reihe geschehen, die Kopieranweisungen aber überlappen. Dazu werden alle Operationen verschiedenen Streams zugeordnet, aber explizite Synchronisierung verhindert eine verfrühte Addition oder Kopie.

.....

Unified (Virtual) Memory

Das manuelle Kopieren von Speicher ist oft sehr mühselig. Daher hat Nvidia den Grafiktreiber soweit entwickelt, dass dies mittlerweile weitestgehend automatisch möglich ist. Dafür wurde eine spezielle Art der virtuellen Speicherverwaltung implementiert, bei der CPU und GPU Speicher in den selben Adressraum gemappt wird. Man erstellt im Programm dazu lediglich mittles

```
float *d_A; cudaMallocManaged(&d_A, size);
```

einen einzelnen Pointer, der sowohl von GPU als auch von CPU aus erreichbar ist. Alternativ lässt sich das Keyword __managed__ für globale Arrays (auch in Verbindung mit __device__) benutzen.

Begriffsklärung:

Da es sich um einen einzelnen Pointer handelt, ist der Begriff *Unified Memory* theoretisch gerechtfertigt. Es handelt sich jedoch nur um eine Art von virtueller Speicherverwaltung. „Virtuell“ bezieht sich dabei auf den Speicher, nicht auf die Verwaltung. Bei *Unified Memory* spricht man üblicherweise von Systemen, in denen sich CPU und GPU physisch den selben Speicher teilen. Dies ist z.B. oft bei Einplatinencomputern der Fall. Man spricht von einer *Unified Memory Architecture (UMA)*. Geht es bei dem Zugriff um die parallele Nutzung innerhalb eines symmetrischen Multiprozessorsystems, so spricht man von *Uniform Memory Access*, ebenfalls *UMA*. Aus offensichtlichen Gründen wird hierbei auch von *Shared Memory* gesprochen. Dies sollte aber auf keinen Fall mit CUDA Shared Memory verwechselt werden.

Wird solch ein „managed memory“ auf einer prä-Pascal Karte alloziert, so handelt es sich dabei komplett und zu jedem Zeitpunkt um GPU Speicher. Versucht die CPU nun auf diese Elemente zuzugreifen, tritt ein *page fault* auf. Durch verborgenen Prozesse (ausgeführt durch den Grafiktreiber) werden nun die benötigten Daten vom GPU in den CPU Speicher in Einheiten von Memory Pages (4kB) kopiert. Der Treiber garantiert Kohärenz zwischen den Speichern, beachtet aber keine Dataraces.

Auf einer Pascal oder Volta Karte wird der Speicher dort alloziert, wo er zuerst initialisiert wird. Diese moderneren GPUs beherrschen ebenfalls Pagefaulting. Sollte die Größe der Allozierung den Speicher der GPU überschreiten wird also kein Fehler geworfen. Da fast nie die komplette Datenmenge im Speicher der GPU vorhanden sein muss, braucht der Grafiktreiber lediglich die benötigten Daten zu kopieren. Das Synchronisieren der Threads läuft automatisch. Natürlich gibt es durch diese Kopien einen unnötigen Overhead. Außerdem sind GPUs eher für große Datenmengen optimiert. Daher sollte man bei dieser Methode von einem Performanceverlust von etwa 10 bis 20% ausgehen.

Da die GPU auf Daten normalerweise sehr oft operiert, ohne dass die CPU diese benötigt, lässt sich der Overhead minimieren:

- Initialisieren der Daten in einem eigenen Kernel anstatt im Speicher der CPU
- Mehrmaliges Ausführen des Kernels und Analysieren der mittleren Laufzeit mit nvprof
- Explizites Kopieren: Um die *Page Migration Engine* explizit zu benutzen existiert der Befehl cudaMemPrefetchAsync(d_A, size, deviceID, stream). Ein Wert von -1 bei der deviceID bezeichnet die CPU.

Genauere Informationen unter https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge8dc9199943d421bc8bc7f473df12e42.

Außerdem kann mittels cudaMemAdvise(d_A, size, advice, deviceID) das Verhalten der Engine geführt werden. Mehr unter https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#group__CUDART__MEMORY_1ge37112fc1ac88d0f6bab7a945e48760a.

3.7. Datentypen

Auf GPUs existiert ein 16 Bit Datentyp mit halber Präzision. Benutzt man die in CUDA eingebauten arithmetischen Funktionen, kann man die Performance dadurch weiter verbessern.

```
#include<cuda_fp16.h>
#include "fp16_conversion.h"

__global__
void haxpy(int n, half a, const half *x, half *y)
{
    int start = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x;

  int n2 = n/2;
  half2 *x2 = (half2*)x, *y2 = (half2*)y;

  for (int i = start; i < n2; i+= stride)
      y2[i] = __hfma2(__halves2half2(a, a), x2[i], y2[i]);


  if (start == 0 && (n%2))
      y[n-1] = __hfma(a, x[n-1], y[n-1]);
}

```

Codebeispiel 3.16: Half Precision

Die benutzten Funktionen verlangen den Datentyp half2, also eine Kombination von zwei half Werten.

Eingebaute Funktionen wären z.B. __hdiv, __hadd oder __hmul. Eine Auflistung aller Funktio-

nen befindet sich in Kapitel 1.1 der Dokumentation der CUDA Math API. [6]

Der Header *cuda_ fp16.h* befindet sich im üblichen include-Ordner. *fp16_ conversion.h* benötigt man jedoch aus dem github: `https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/mixed-precision/fp16_conversion.h`

Diese Datei enthält Konvertierungsfunktionen für float nach half.

3.8. Atomic Operations und Reduktionen

Eine Reduktion ist eine Operation, die die Dimensionalität eines Objekts verringert. Üblicherweise sind damit Operationen gemeint, die ein Array beliebiger Größe auf ein einfaches Skalar verringern. Beispiele dafür sind die Maximumsreduktion (also das Auffinden des größten Wertes), die Minimumsreduktion, die Summenreduktion (also die Summe aller Elemente) oder das Produkt aller Elemente.

Dies impliziert, dass viele Threads gleichzeitig den selben Wert aktualisieren müssen, z.B. beim Addieren einer Zahl zu einem globalen Zähler. An dieser Stelle tritt ein sogenanntes Datarace auf: Ein Thread liest einen Wert aus dem Speicher, addiert und legt die Variable wieder im Speicher ab und überschreibt damit einen Wert der währenddessen von anderen Threads bereits bearbeitet wurde bzw. bearbeitet wird. Um diese Zugriffe zu serialisieren, stellt CUDA sogenannte Atomic Operations zur Verfügung, also Operationen, die garantiert seriell ablaufen. Beispielsweise int atomicAdd(int* address, int val) addiert auf eine Variable bei der Speicheradresse address den Wert val. Eine Liste befindet sich im Anhang B.12 des CUDA Programming Guides. [7]

Diese Form der Implementierung ist sehr langsam und eher zum Debuggen gedacht. Eine parallele Methode ist erforderlich. Im Prinzip nimmt zu Beginn jeder Thread zwei Werte und führt die Operation aus. Der nächste Thread nimmt abermals zwei Werte ohne Überlapp zum Vorherigen. Nach einem Schritt hat sich die Anzahl der zu reduzierenden Elemente halbiert. Im nächsten Schritt wird mit der halben Zahl der Threads abermals halbiert, bis die Reduktion bereit steht.

Das folgende Beispiel zeigt eine einfache Implementierung der Summenreduktion:

```
__global__ void reduction(int *data)
{
  uint tid = threadIdx.x;
  uint i = blockIdx.x*(blockDim.x*2)+threadIdx.x;

  extern __shared__ int sm[];
  sm[tid] = data[i] + data[i + blockDim.x];

  for (uint stride = blockDim.x/2; stride > 32; stride >>= 1)
  {
    __syncthreads();
    if(tid < stride) sm[tid] += sm[tid + stride];
  }
  if (tid < 32)
    {
    sm[tid] += sm[tid + 32];
    sm[tid] += sm[tid + 16];
    sm[tid] += sm[tid +  8];
    sm[tid] += sm[tid +  4];
    sm[tid] += sm[tid +  2];
    sm[tid] += sm[tid +  1];
  }

  if(tid == 0) data[blockIdx.x] = sm[0];
}

  ...
  uint maxThreads = ...;
  uint threads = (n < maxThreads) ? n : maxThreads;
  uint blocks = n/threads;

  uint smSize = threads*sizeof(int);

  reduction<<<blocks, threads, smSize>>>(data);

  uint toDo = 0;
  if (blocks > 1) toDo = 1 + blocks/maxThreads;
  for(uint i = 0; i < toDo; ++i)
  {
    threads = (blocks < maxThreads) ? blocks : maxThreads;
    blocks /= threads;
```

```
42          reduction<<<blocks, threads, smSize>>>(d);
        }
44      ...
```

Codebeispiel 3.17: Reduktion

Verschiedene HPC Librarys bieten Implementierungen der gängigsten Reduktionen an, z.B. THRUST oder MAGMA.

3.9. Dynamischer Parallelismus

```
    ...
2       kernel_parent<<<1,1>>>(device, &N);
    ...
4
__global__ void kernel_parent(float* data, uint* N)
6   {
        uint newN = *N/2;
8
    cudaStream_t stream1;
10  cudaStreamCreateWithFlags(&stream1, cudaStreamDefault);

12  float *device1;
    cudaMalloc(&device1, newN*sizeof(float));
14  cudaMemcpyAsync(device1, data, newN*sizeof(float),
      cudaMemcpyDeviceToDevice, stream1);
16
    kernel_child <<<1, 1, 0, stream1>>>(device1, &newN);
18
    cudaStream_t stream2;
20  cudaStreamCreateWithFlags(&stream2, cudaStreamDefault);
    float *device2;
22  cudaMalloc(&device2, newN*sizeof(float));
    cudaMemcpyAsync(device2, data+*N/2, newN*sizeof(float),
24    cudaMemcpyDeviceToDevice, stream2);

26  kernel_child <<<1, 1, 0, stream2>>>(device2, &newN);
}
```

```
28
    __global__ void kernel_child(float* data, uint *N)
30  {
      uint newN = *N/2;
32
      cudaStream_t stream1;
34    cudaStreamCreateWithFlags(&stream1, cudaStreamDefault);

36    float *device1;
      cudaMalloc(&device1, newN*sizeof(float));
38    cudaMemcpyAsync(device1, data, newN*sizeof(float),
        cudaMemcpyDeviceToDevice, stream1);
40
      kernel_child <<<1, 1, 0, stream1>>>(device1, &newN);
42
      cudaStream_t stream2;
44    cudaStreamCreateWithFlags(&stream2, cudaStreamDefault);

46    float *device2;
      cudaMalloc(&device2, newN*sizeof(float));
48    cudaMemcpyAsync(device2, data+*N/2, newN*sizeof(float),
        cudaMemcpyDeviceToDevice, stream2);
50
      kernel_child <<<1, 1, 0, stream2>>>(device2, &newN);
52  }
```

Codebeispiel 3.18: Dynamischer Parallelismus

Kompilieren: nvcc dynpar.cu −arch=compute_61 −code=sm_61 −rdc=true −lcudadevrt

3.10. CUDA-Toolkit

Baumstruktur

NVCC

Der Nvidia CUDA Compiler (nvcc) wird benutzt um sowohl Host- als auch Devicecode zu kompilieren. Es handelt sich dabei um eine Obermenge des g++. Es lässt sich bis zum C++14 Standard alles in C oder C++ kompilieren. Soll explizit CUDA Code kompiliert werden, so muss die Quelldatei die Dateiendung .cu tragen. Andernfalls wird der Code als gewöhnliches C++ kompiliert und der Compiler liefert Fehler, sobald er auf Devicecode oder auf einen Aufruf der Runtime Library stößt.

Der Compiler wird wie gewohnt gestartet:

nvcc <name>.cu

Der Compiler kompiliert nun getrennt Hostcode wie gewohnt mittels gcc/g++ (clang unter MacOS), Devicecode aber in eine assemblerartige Sprache namens *Nvidia Parallel Thread Execution* (NVPTX). Aus dieser kann Maschinencode erstellt werden, der für die GPU lesbar ist. Der Compiler erstell ein sogenanntes *fatbinary*, kompiliert also Device- und Hostcode in die selbe ausführbare Datei. Um dies zu verhindern, kann die Option −cubin verwendet werden. Damit wird der Devicecode in ein eigenes Binary kompiliert, ein sogenanntes *cubin*. Im Handbuch des nvcc [20] befindet sich eine Liste der Compilerflags. Diese sind in den wesentlichen Punkten kompatibel zu g++ mit Ausahme einiger spezieller Modi für Devicecode.

Die wichtigste Ausnahme ist: −arch=compute_35 −code=sm_35

Dieses lässt den Code explizit für eine compute capability kompilieren, in diesem Fall 3.5. Damit lassen sich Features dieser compute capability aktivieren. Allerdings ist der Code auf älteren Karten dann nicht mehr ausführbar (falls modernere Features verwendet wurden). ”Arch” (Architecture) bezeichnet dabei das Chipdesign, ”Code” die Software. Gibt man mehrere Optionen an (getrennt mit Komma) an, so werden Cubins für alle Architekturen erstellt und in einem einzelnen Fatbinary verschmolzen. Beim Ausführen wir dann automatisch die richtige Architektur erkannt.

Mit –c lassen sich wie gewohnt Objektdateien (.o) und damit statische und dynamische Librarys (–shared) erstellen.

Der Compiler inkludiert automatisch einige Header z.B. cuda.h oder math.h, sowohl für Host- als auch für Devicecode und linkt selbstständig mit den richtigen Bibliotheken. Für weitere Bibliotheken können selbstverständlich die Compilerflags –I und -–L sowie der Linker, z.B. für –lgomp benutzt werden. Soll der zugrunde liegende C++-Compiler konfiguriert werden, wird die Option –Xcompiler benötigt. Was nach dieser Option in Anführungszeichen folgt, wird dem C++-Compiler exakt so mitgegeben, z.B. –Xcompiler "–Wall –fopenmp –std=c++17".

Das Verhalten des Compilers kann über verschiedene Umgebungsvariablen gesteuert werden. Eine Liste befindet sich im Anhang J des CUDA Programming Guides. [7]

Eine Objektdatei oder eine Bibliothek kann auch zusammen mit einer gewöhnlichen C++ Datei (.cpp) gelinkt werden. Dazu muss aber unter Umständen diese Datei die entsprechenden Header (cuda.h, cuda_runtime.h) inkludieren und mit -lcudart linken. Dann kann auch ein anderer C-Compiler verwendet werden. Dies kann man nutzen, um Devicecode in eigene Module auszulagern und diesen damit von Hostcode strikt zu trennen. Dadurch lassen sich auch andere Compiler und deren Features mit Cuda kombinieren, z.B. der Intel- oder PGI-Compiler.

cuda-memcheck

nvprof

Nvidia Nsight

nvprune

nvdisasm und cuobjdump

Einführung in OpenCL

4. Einführung in OpenCL

Open Computing Language (OpenCL) ist eine freie open-source Programmierschnittstelle für paralleles Rechnen, die von Apple im Jahr 2009 entwickelt wurde. Heute liegt diese in Version 2.2 vor und wird von der Khronos Gruppe entwickelt, der sich mittlerweile über 200 Firmen angeschlossen haben. Das Ziel hinter dem Projekt ist es, eine low-level Programmiersprache und Schnittstelle zu schaffen, mit denen sich maximal inhomogene Parallelrechner (CPUs, GPUs, Grafikchips, FPGAs, Beschleuniger, ...) programmieren lassen und dabei Plattformunabhängigkeit gewährleisten. Ein Gerät benötigt dafür eine Hardwareimplementierung und der Hersteller muss eine OpenCL Implementierung in Software bereitstellen (z.B. AMD, Intel, ...). Im Nvidia Toolkit ist eine OpenCL Installation enthalten, in Hardware aber nur für Version 1.2 implementiert. C- und C++-APIs sind gewöhnliche Librarys, die unter Linux über die Paketquellen installiert werden können.

Die Khronos Gruppe entwickelt ebenfalls die Grafikbibliothek OpenGL und Nachfolger Vulkan, für die ähnliche Prinzipien gelten.

4.1. Begriffe

Obwohl Nvidia ebenfalls Teil der Khronos Gruppe ist, haben sich in der Fachsprache hier andere Begriffe etabliert. Tabelle 4.1 zeigt typische OpenCL Begriffe und ihre Entsprechung in CUDA. Technisch gesehen sind sich AMD und Nvidia Grafikkarten ähnlich. Allerdings nennt AMD die Warps Wavefronts und deren Größe ist abhängig vom Chip entweder 32 oder 64. Demnach sollte bei entsprechender Größe die Local Work Size, also die Größe einer Workgroup, ein Vielfaches von 32 oder 64 sein.

Diese Einführung beschränkt sich auf das Programmieren von GPUs. Die dafür verwendeten Befehle sind CUDA sehr ähnlich und tragen lediglich andere Namen. Tabelle 4.1 zeigt eine

OpenCL	CUDA
Workitem	Thread
Workgroup	Block
Global Memory	Global Memory
Constant Memory	Constant Memory
Local Memory	Shared Memory
Private Memory	Local Memory
Platform	*keine*
Device	Device
Context	*keine*
Command Queue	Stream
Kernel	Kernel
Event	Event
Global Work Size	Grid Size
Local Work Size	Block Size
Work Dimension	Grid/Block Dimension

Tabelle 4.1.: Typische OpenCL Begriffe und ihre Entsprechung in CUDA

Auflistung der wichtigsten.

Für die Nvidia Produktlinien Geforce, Quadro und Tesla existieren bei AMD die Entsprechungen Radeon, Radeon Pro und Radeon Instinct.

4.2. Programmiermodell

Im Gegensatz zu CUDA fordert OpenCL eine strikte Trennung von Host- und Devicecode in verschiedenen Dateien. Das Hostprogramm wird mittels der OpenCL C-API programmiert (oder C++ Wrapper), bei der es sich um eine gewöhnliche C-Library handelt. Dieses Hostprogramm wird mit einem beliebigen C- oder C++-Compiler übersetzt und unterstützt damit auch die modernste Features beider Programmiersprachen.

Das Deviceprogramm, also die Kernel liegen in einer oder mehrerer seperater Dateien vor, die als String vom Hostprogramm eingelesen werden. Dies hat den Vorteil, dass zu Compilezeit der Inhalt des Programms nicht bekannt sein muss und damit auch nicht verändert wird. Erst zur Laufzeit wird unsichtbar für den Nutzer der OpenCL Compiler gerufen und das OpenCL Programm kompiliert und ausgeführt (Just-In-Time Compiler, JIT). Bei einer Änderung eines

OpenCL	CUDA
__kernel, kernel	__global__
__constant, constant	__constant__
__global, global	*automatisch*
__private, private	*automatisch*
__local, local	__shared__
get_global_id(0)	blockIdx.x*blockdim.x + threadIdx.x
get_global_id(1)	blockIdx.y*blockdim.y + threadIdx.y
get_global_id(2)	blockIdx.z*blockdim.z + threadIdx.z
get_local_id(0)	threadIdx.x
get_local_id(1)	threadIdx.y
get_local_id(2)	threadIdx.z
get_local_size(0)	blockdim.x
get_local_size(1)	blockdim.y
get_local_size(2)	blockdim.z
get_global_size(0)	griddim.x
get_global_size(1)	griddim.y
get_global_size(2)	griddim.z
clCreateBuffer (...)	cudaMalloc(...)
clEnqueueWriteBuffer (...)	cudaMemcpy(Async)(..., cudaMemcpyHostToDevice)
clEnqueueReadBuffer(...)	cudaMemcpy(Async)(..., cudaMemcpyDeviceToHost)
clEnqueueCopyBuffer(...)	cudaMemcpy(Async)(..., cudaMemcpyDeviceToDevice)
clEnqueueNDRangeKernel(..., <Kernel>, ...), clSetKernelArg (...)	< kernelcall ><<<...>>>(...)
barrier (CLK_LOCAL_MEM_FENCE)	__syncthreads()
barrier (CLK_GLOBAL_MEM_FENCE)	__syncthreads()
barrier (CLK_IMAGE_MEM_FENCE)[1]	*keine*

Tabelle 4.2.: Typische OpenCL Befehle/Keywords und ihre Entsprechung in CUDA

Kernel muss das Hostprogramm also nicht noch ein weiteres Mal kompiliert werden.

Devicecode wird in der eigentlichen Programmiersprache OpenCL C geschrieben, die sich im Wesentlichen an C99 orientiert. Teilweise wurde auch C++ unter dem Namen OpenCL C++ implementiert (siehe 4.7). Es folgt eine Auflistung der wichtigsten Einschränkungen:

- keine C99-Standard Header, Ersatz ist der Vorrat von built-in-Funktionen.
- Kernel-Zeigerargumente müssen mit __global, __constant, oder __local qualifiziert werden.
- nur __constant-Zeiger dürfen __constant-Zeigern zugewiesen werden.
- keine Zeiger auf Funktionen.
- keine Bitfelder.

- keine VLAs (variable length arrays).
- weder Makros noch Funktionen mit variabler Argumentzahl (außer enqueue_kernel).
- keine auto und register Speicherklassen.
- keine rekursiven Funktionen.
- Kernel sind immer Prozeduren mit void-Ergebnis.
- keine Kernel-Argumente mit den Typen bool, half, size_t, ptrdiff_t, intptr_t, uintptr_t, sowie Strukturen und Unions mit einer dieser Komponenten.
- keine Arithmetik für half, nur Speicherformat.
- irreduzible Anweisungsgruppen (z.B. Sprünge in Schleifen) sind implementation-defined.
- const, restrict und volatile sind erlaubt, aber für Images verboten.
- event_t darf kein Kernel-Argument sein.
- event_t darf nicht zusammen mit __local, __constant and __global verwendet werden.

Platformen

Da mehrere verschiedene Implementierungen von OpenCL existieren, muss zunächst eine Platform ausgewählt werden. Die Khronos Gruppe aktualisiert stetig die Liste aller Platformen: https://www.khronos.org/conformance/adopters/conformant-products/opencl

```
    cl_platform_id platform_id = NULL;
2   uint ret_num_platforms;
    clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
4
```

Codebeispiel 4.1: Platformabfrage

platform_id kann bei mehreren Platformen ein Array von IDs sein. In diesem Fall muss im ersten Argument angegeben werden, nach wie vielen Platformen gesucht werden soll.

Die meisten API Aufrufe geben einen Fehlercode zurück, der in einer Tabelle nachgeschlagen werden kann: https://streamhpc.com/blog/2013-04-28/opencl-error-codes/

Im Folgenden werden nur die OpenCL eigenen Datentypen verwendet (siehe 4.3).

Devices

Jede Platform stellt OpenCL für ein oder auch mehrere Geräte zur Verfügung. Im nächsten Schritt müssen also die entsprechenden Devices den gefundenen Platformen zugeordnet werden.

```
cl_device_id device_id = NULL;
uint ret_num_devices;
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
  &device_id, &ret_num_devices);
```

Codebeispiel 4.2: Deviceabfrage

Auch hier kann device_id ein Vektor sein. Die zweite Eingabe bezeichnet als Bitfeld die Device Typen, die abgefragt werden sollen. Mehrere Typen können mit dem Operator | abgetrennt werden. Für den Typ des Geräts existieren folgende Möglichkeiten:

- CL_DEVICE_TYPE_CPU
- CL_DEVICE_TYPE_GPU
- CL_DEVICE_TYPE_ACCELERATOR
- CL_DEVICE_TYPE_DEFAULT
- CL_DEVICE_TYPE_ALL

Kontexte

Ein Kontext ist ein Handle für das OpenCL Programm. Dieser beinhaltet Speicherobjekte, Programme und Command Queues. Folglich müssen diese also einem neu erstellten Kontext

zugeordnet werden. Abbildung 4.1 zeigt die Zusammenhänge sämtlicher OpenCL Klassen [21] basierend auf der Unified Modelling Language (UML). [36]

```
    cl_int ret;
2   cl_context context = clCreateContext(NULL, 1, &device_id,
      NULL, NULL, &ret);
4
```

Codebeispiel 4.3: Kontexte

Sollte eine Funktion über einen eigenen Rückgabewert verfügen, so ist das letzte Argument üblicherweise ein Pointer auf einen Fehlercode.

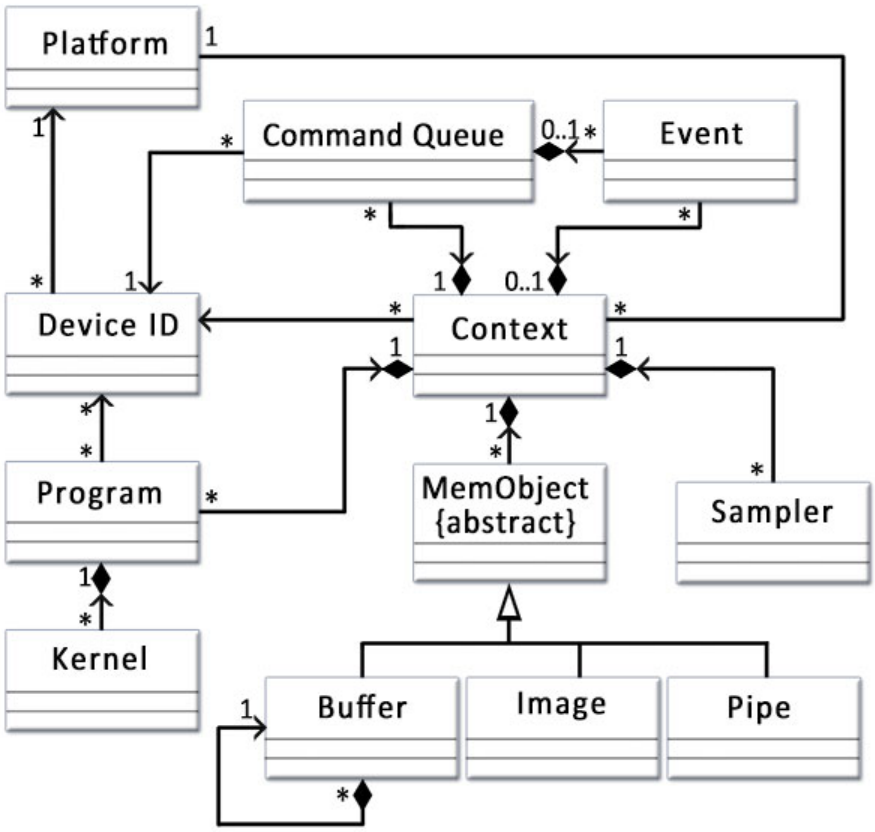

Abbildung 4.1.: Zusammenhänge sämtlicher OpenCL Klassen

Programme

Als nächstes muss das OpenCL Programm als String gelesen, zur Laufzeit kompiliert und dem Kontext hinzugefügt werden.

```
FILE *fp;
char *source_str;
size_t source_size;
fp = fopen("cl_vecadd.cl", "r");
if(!fp)
{
  fprintf(stderr, "Failed to load the kernel!\n");
  exit(1);
}
fseek(fp, 0, SEEK_END);
size_t s = ftell(fp);
fseek(fp, 0, SEEK_SET);
source_str = (char*)malloc(s);
source_size = fread(source_str, 1, s, fp);
fclose(fp);

cl_program program = clCreateProgramWithSource(context, 1,
  (const char **)&source_str, (const size_t *)&source_size, &ret);
```

Codebeispiel 4.4: OpenCL Programm

Mittels clBuildProgram(program, 1, &device_id, "−D name=definition −I dir", NULL, NULL) lassen sich ähnlich dem C-Compiler dem OpenCl Compiler zusätliche Informationen mitgeben (siehe auch https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clBuildProgram.html). Ebenso muss man hier eine Liste der Devices übergeben, für die der Code kompiliert werden soll. So lässt sich der Include-Pfad erweitern. Außerdem können mit −D Makros gesetzt werden. Eine Auflistung voreingestellter Makros findet sich in Kapitel 4.3.

Das vorletzte Argument kann einen Pointer auf eine Funktion enthalten, die nach erfolgreichem Kompilieren ausgeführt werden soll (Callback). Das letzte Argument kann die Argumente dieser Funktion beinhalten.

Da an dieser Stelle erst das eigentliche OpenCL Programm kompiliert wird, muss nun der Output des OpenCL Compilers abgefragt werden.

```
    if(ret != 0){
2     size_t sz; char * bf;

4     clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_STATUS,
        0, NULL, &sz);
6     bf = (char*)malloc((sz+1) * sizeof(char));
      if(bf)
8     {
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_STATUS,
10        sz+1, bf, NULL);
        bf[sz] = 0;
12      fprintf(stderr, "\n%s\n", bf);
        free(bf);
14    }

16    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_OPTIONS,
        0, NULL, &sz);
18    bf = (char*)malloc((sz+1) * sizeof(char));
      if(bf)
20    {
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_OPTIONS,
22        sz+1, bf, NULL);
        bf[sz] = 0;
24      fprintf(stderr, "\n%s\n", bf);
        free(bf);
26    }

28    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
        0, NULL, &sz);
30    bf = (char*)malloc((sz+1) * sizeof(char));
      if(bf)
32    {
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
34        sz+1, bf, NULL);
        bf[sz] = 0;
36      fprintf(stderr, "\n%s\n", bf);
        free(bf);
38    } }
```

Codebeispiel 4.5: Fehlerabfrage OpenCL Compiler

Kernel

Danach kann das eigentliche Kernel erstellt werden. Dazu muss der Name der Funktion als String übergeben werden. Das Programmieren der Kernels funktioniert ähnlich wie in CUDA.

```
__kernel void vecadd(__constant float *x, __constant float* y, __global
float *res, const int size)
{
    int i = get_global_id(0);
    if(i < size)
      res[i] = x[i] + y[i];
}
```

Codebeispiel 4.6: Kerneldefinition

Kernel müssen mit dem Keyword __kernel deklariert. Der Rückgabewert ist immer void. Jedes Objekt liegt per Default im private Memory des Workitems. Arrays müssen also mit dem entsprechenden Keyword deklariert werden, __global für global Memory, __constant für constant Memory und __local für local Memory. Die Unterstriche können weggelassen werden.

Eine zusätlich definierte Funktion kann vom Kernel, also vom Device aus, wie aus C gewohnt ausgeführt werden.

Für das Kernel müssen wie gewohnt Buffer erstellt und Speicher kopiert werden. Zudem müssen die Kernelargumente explizit gesetzt werden.

```
cl_mem d_x  = clCreateBuffer(context, CL_MEM_READ_ONLY,
  size*sizeof(cl_float), NULL, &ret);
cl_mem d_y  = clCreateBuffer(context, CL_MEM_READ_ONLY,
  size*sizeof(cl_float), NULL, &ret);
cl_mem d_res = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
  size*sizeof(cl_float), NULL, &ret);
cl_kernel kernel_vecadd = clCreateKernel(program, "vecadd", &ret);
size_t narg = 0;
clSetKernelArg(kernel_vecadd, narg++, sizeof(cl_mem),  (void *)&d_x);
clSetKernelArg(kernel_vecadd, narg++, sizeof(cl_mem),  (void *)&d_y);
clSetKernelArg(kernel_vecadd, narg++, sizeof(cl_mem),  (void *)&d_res);
clSetKernelArg(kernel_vecadd, narg++, sizeof(cl_uint), &size);
```

Codebeispiel 4.7: Kernelaufruf

Ein Buffer kann in verschiedenen Modi erstellt werden (4.3).

CL_MEM_READ_WRITE	Zum Lesen und zum Schreiben (default)
CL_MEM_WRITE_ONLY	Das Objekt wird vom Kernel nicht gelesen
CL_MEM_READ_ONLY	Das Objekt wird vom Kernel nicht geschrieben
CL_MEM_USE_HOST_PTR	Erstellt Speicher im Devicememory aus Hostarray. Erstellen von verschiedenen Buffern aus dem selben Pointer ist nicht definiert.
CL_MEM_ALLOC_HOST_PTR	Wie CL_MEM_USE_HOST_PTR, aber mit automatischer Allozierung
CL_MEM_COPY_HOST_PTR	Wie CL_MEM_USE_HOST_PTR, aber mit automatischer Kopie

Tabelle 4.3.: Buffermodi

Das vorletzte Argument ist ein Pointer auf den Hostspeicher.

Mehrere Flags können grundsätzlich über eine Abtrennung mit | gleichzeitig gesetzt werden. Mit der Angabe von CL_TRUE wird der Host blockiert (CL_FALSE entspricht also \licudaMemcpyAsync(...)“). Die Angabe danach gibt einen Offset auf der Speicheradresse an.

Command Queues

Statt Streams existieren in OpenCL Command Queues. Jede Speicheranweisung (Kopieren, Lesen) muss in eine Command Queue eingereiht werden. Zum Schluss wird die Warteschlange mit dem Kernel ausgeführt und erzeugt dabei ein Event. Das Lesen des Speichers erfolgt, sobald dieses Event stattgefunden hat. Am Ende wird der Speicher freigegeben.

```
    const cl_queue_properties  props = ...;
2   cl_command_queue command_queue = clCreateCommandQueueWithProperties(context,
    device_id, &props, &ret);

4   clEnqueueWriteBuffer(command_queue, d_x, CL_TRUE, 0,
      size*sizeof(cl_float), h_x, 0, NULL, NULL);
6   clEnqueueWriteBuffer(command_queue, d_y, CL_TRUE, 0,
      size*sizeof(cl_float), h_y, 0, NULL, NULL);
8

10
```

```
    size_t global_item_size = ... //Zahl der Workitems gesamt
12  size_t local_item_size = ... //Größe der Workgroup
    cl_event vecadd_event;
14  clEnqueueNDRangeKernel(command_queue, kernel_vecadd, 1, NULL,
      &global_item_size, &local_item_size, 0, NULL, &vecadd_event);
16
    clEnqueueReadBuffer(command_queue, d_res, CL_TRUE, 0,
18    size*sizeof(cl_float), h_res, 1, &vecadd_event, NULL);

20  /* clFlush(command_queue); */
    clFinish(command_queue);
22
    clReleaseKernel(kernel_vecadd);
24  clReleaseProgram(program);

26  clReleaseMemObject(d_x);
    clReleaseMemObject(d_y);
28  clReleaseMemObject(d_res);

30  clReleaseCommandQueue(command_queue);
    clReleaseContext(context);
32
    free(...);
34
```

Codebeispiel 4.8: Command Queues und Clean-Up

props bezeichnet eine Liste verschiedener Flags, mit denen die Command Queue erstellt werden kann. Es existieren zwei Möglichkeiten:

- CL_QUEUE_PROPERTIES: ein Bitfeld, dass eine Kombination folgender Werte sein kann:
  - CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE: ermöglicht Concurrency auf dem Device in der selben Command Queue
  - CL_QUEUE_PROFILING_ENABLE: ermöglicht Profiling Kommandos
  - CL_QUEUE_ON_DEVICE: erstellt eine Device Queue (kann nur mit CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE verwendet werden)
  - CL_QUEUE_ON_DEVICE_DEFAULT: erstellt die default Device Queue (nur eine pro

Kontext) (kann nur mit CL_QUEUE_ON_DEVICE verwendet werden)

- CL_QUEUE_SIZE: ein cl_uint, der die Größe der Device Queue in Bytes angibt (kann nur mit CL_QUEUE_ON_DEVICE verwendet werden)

In früheren Versionen existierte die Funktion clCreateCommandQueue, wird jedoch mittlerweile als veraltet angesehen.

Das letzte Argument von clEnqueueNDRangeKernel, clEnqueueReadBuffer und clEnqueueWriteBuffer enthält einen Pointer auf ein Event. Das Event hat stattgefunden, sobald die Funktion ausgeführt wurde. Das vorletzte Argument enthält eine Liste von Events. Das drittletzte Argument gibt an, auf wie viele dieser Events gewartet werden muss, bis die Funktion ausgeführt werden soll. Da sixh die Kommandos in der selben Command Queue befinden, werden sie ohnehin in Reihe ausgeführt werden. Wird die Command Queue jedoch mit der Eigenschaft CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE erstellt, so überlappen die Aktionen möglicherweise (Concurrency) und müssen explizit synchronisiert werden. Alternativ lassen sich für verschiedene Kontexte und Devices mehrere Command Queues erstellen und so parallel bearbeiten.

Jeder API Aufruf erfolgt also grundsätzlich asynchron. clFinish bildet eine Barriere für die CPU bis die entsprechende Command Queue ausgeführt wurde. Für Thread-paralleles Programmieren der CPU existiert der Befehl clFlush, der eine Barriere bildet, bis eine bestimmte Command Queue aufgebaut wurde.

4.3. Datentypen und Makros

Das OpenCL API definiert sich einige Datentypen um Plattformunabhängigkeit zu gewährleisten. Beispielsweise garantiert cl_int, dass es sich dabei unabhängig von Compiler und Betriebssystem um eine 32bit Ganzzahl handelt.

Datentypen:

- Skalare:
  https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/scalarDataTypes.html

- Vektor:
  https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/vectorDataTypes.html

- Abstrakte:
  https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/abstractDataTypes.html

- Reservierte:
  https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/reservedDataTypes.html

- Sonstige:
  https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/otherDataTypes.html

Alle Präprozessordirektiven von C99 werden unterstützt. Einige weitere wurden zusätzlich definiert:
https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/preprocessorDirectives.html

4.4. Images

Das Image Format ist eine Datenstruktur, die speziell auf Bilformate optimiert wurde.

```
    cl_mem clCreateImage(
2     cl_context context,
      cl_mem_flags flags,
4     const cl_image_format *image_format,
      const cl_image_desc *image_desc,
6     void *host_ptr,
      cl_int *errcode_ret)
8
```

Codebeispiel 4.9: OpenCL Images

Das Erstellen funktionier ähnlich zu einem gewöhnlichen Speicherobjekt. Allerdings handelt es sich dabei um eine Datenstruktur, deren genauer Inhalt angegeben werden muss. image_format

enthält Informationen über das Datenformat des Bildes: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/cl_image_format.html

Bei einem CL_ FLOAT/RGB-Bild besteht jedes Element der Datenstruktur also aus vier float Werten, von denen jede ein Pixel des Bildes beschreibt, also Rotwert, Grünwert, Blauwert und Alphachannel.

cl_image_dsc enthält Informationen über die Struktur des Bildes, z.B. Höhe und Breite in Pixel: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/cl_image_desc.html

4.5. Pipes

Das Pipe Objekt exisitiert erst seit OpenCl 2.0 und bezeichnet einen globalen Memorybuffer der kontroliert gelesen und geschrieben werden kann. Pipes können vom Host nur zur Übergabe an ein Kernel benutzt werden. Im Device Code können Pipes nur über built-in Funktionen verändert werden: https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/pipeFunctions.html

4.6. C++ API

Es existiert eine C++ Wrapper API [15]. Das Beispiel der Vektoraddition lässt sich somit in C++ umschreiben:

```
    std::vector<cl::Platform> platformList;
2   cl::Platform::get(&platformList);
    cl_context_properties cprops[] = {
4     CL_CONTEXT_PLATFORM, (cl_context_properties)(platformList[0])(), 0};

6   cl::Context context(CL_DEVICE_TYPE_GPU, cprops);

8   std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

10  cl::Program::Sources sources(1, std::make_pair(source_str, 0));
    cl::Program program(context, sources);
12  program.build(devices);
```

```
14    cl::Buffer d_x = cl::Buffer(context,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
16      size*sizeof(cl_float), (void*)&h_x[0]);

18    cl::Buffer d_y = cl::Buffer(context,
        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
20      size*sizeof(cl_float), (void*)&h_y[0]);

22    cl::Buffer d_res = cl::Buffer(context,
        CL_MEM_WRITE_ONLY, size*sizeof(cl_float));
24
      cl::Kernel kernel(program, "vecadd");
26    size_t narg = 0;
      kernel.setArg(narg++, d_x);
28    kernel.setArg(narg++, d_y);
      kernel.setArg(narg++, d_res);
30    kernel.setArg(narg++, size);

32    cl::CommandQueue queue(context, devices[0], 0);

34    queue.enqueueNDRangeKernel(
        kernel,
36      cl::NullRange,
        cl::NDRange(size),
38      cl::NullRange);

40    cl_float *h_res = (cl_float*)queue.enqueueMapBuffer(d_res,
        CL_TRUE, CL_MAP_READ, 0, size*sizeof(cl_float));
42
      //Ausgabe auf Host
44
      queue.enqueueUnmapMemObject(d_res, (void*)h_res);
46
```

Codebeispiel 4.10: OpenCL C++ API

Die OpenCL Objekte sind nun Instanzen echter C++ Klassen, die sich im Namespace cl befinden. Das cl_mem Objekt nennt sich nun Buffer und wird durch einen expliziten Aufruf eines Konstruktors erstellt. Funktionen, die auf Objekten operieren, sind nun Member-Funktionen der entsprechenden Klasse. Am Ende eines Scopes wird ihr Destruktor automatisch aufgerufen, ein explizites freigeben von Speicher ist daher nicht nötig (außer Ausgabearray auf Host).

4.7. OpenCL C++

Die OpenCL C++ Kernel Language ist seit Version 2.2 verfügbar und enthält eine Untermenge des C++14 Standards, z.B. Lambda Expressions, Templates oder Klassen. Außerdem existiert eine optimierte Implementierung der Standard Template Library, die sich an C++11 orientiert.

Folgende Features wurden bisher noch nicht implementiert:

- Exceptions
- Allocate/Release Memory
- Virtual Functions
- Abstract Classes Function Pointers
- Rekursionen
- goto

Für OpenCL C++ wurde eine eigene Spezifikation angefertigt. [33]

Folgendes Beispiel zeigt eine gewöhnliche Template-Klasse für Matrizen in C++:

```
template<typename T, size_t Rows, size_t Columns>
class matrix
{
   public:
  matrix(){}
  T& operator()(size_t row, size_t col) {return _data[row-1][col-1];}

  constexpr size_t num_rows(){return Rows;}
  constexpr size_t num_columns(){return Columns;}

   private:
  T _data[Rows][Columns];
};
```

Codebeispiel 4.11: Matrix OpenCl C++

Die Addition zweier Matrizen lässt sich durch Überladen des +-Operators implementieren:

```
template<typename T, size_t Rows, size_t Columns>
matrix<T, Rows, Columns>operator+(
const matrix<T, Rows, Columns>& x, const matrix<T, Rows, Columns>& y)
{
  matrix<T, Rows, Columns> tmp;

  for(size_t row = 0; row < Rows; ++row )
  {
    for(size_t column = 0; column < Columns; ++column)
    {
      tmp(row, column) = x(row, column) + y(row, column);
    }
  }

  return tmp;
}
```

Codebeispiel 4.12: Matrixaddition OpenCl C++

Dieser C++ Code lässt sich dank OpenCl C++ exakt so im Devicecode implementieren. In einem Kernel kann dann dieser Code verwendet werden, um auf dem Device eine Matrixaddition auszuführen. Im folgenden Beispiel werden Matrizen addiert, indem jeweils ein float4 zu einer 2×2 Matrix umgewandelt, addiert und danach zurückkonvertiert wird:

```
matrix<float , 2, 2> float4_to_matrix(float4 *in)
{
  matrix<float , 2, 2> m;
  float4 tmp = *in;
  m(1,1) = tmp.s0;
  m(1,2) = tmp.s1;
  m(2,1) = tmp.s2;
  m(2,2) = tmp.s3;

  return m;
}
float4 matrix_to_float4(const matrix<float , 2, 2>& m)
{
  float4 vec;
  vec.s0 = m(1,1) ;
  vec.s1 = m(1,2) ;
```

```
    vec.s2 = m(2,1) ;
    vec.s3 = m(2,2) ;

    return vec;
}

__kernel void add_matrices(float4 *in1, float4 *in2, float4 *result)
{
    size_t idx = get_global_id(0);
    matrix<float, 2, 2> m1 = float4_to_matrix(in1[idx]);
    matrix<float, 2, 2> m2 = float4_to_matrix(in2[idx]);
    result[idx] = matrix_to_float4(m1 + m2);
}
```

Codebeispiel 4.13: OpenCl C++ Kernel

Abgesehen vom Kernel selbst unterscheidet sich dieser Code nicht von gewöhnlichem C++.

5. Radeon Open Compute

Nach dem großen Erfolg von CUDA entschloss sich AMD zur Entwicklung einer eigenen open-source GPU-Plattform für Grafikkarten. Diese ist unter dem Namen *Radeon Open Compute* (ROC) bekannt. Sie spiegelt im Wesentlichen den Programmierstil von CUDA wieder und lässt sich leicht portieren. Die entsprechende Programmiersprache heißt *hip*. Unter `https://github.com/ROCm-Developer-Tools/HIP` stehen die Quelldateien für einen eigenen Build bereit. Für gängige Linux-Betriebssysteme existieren jedoch Pakete für den Paketmanager.

Neben dem AMD Compiler *hipcc* lässt sich im Backend weiterhin *nvcc* verwenden. Folglich ist hip-Code portierbar für AMD und Nvidia GPUs. Entwickler sollen mit diesem Argument motiviert werden, ihre alte Codebasis mit beigefügten Skripten automatisch nach hip zu portieren und dann wahlweise wieder in NVPTX zu kompilieren. Nach diesem Muster wurden die meisten HPC-Librarys von Nvidia bereits portiert und können teilweise als drop-in Ersatz genutzt werden. Wenn man die entsprechenden Pfade in der Compiler-Toolchain ersetzt, lassen sich auch größere Projekte in Sekunden portieren und gleichzeitig in Nvidia- oder AMD-Code übersetzen. Das einfache Beispiel der Vektoraddition aus Kapitel drei könnte mit dem Aufruf `hipify-perl name.cu > name.hip.cpp` portiert werden und würde dann so aussehen:

```
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = hipBlockDim.x * hipBlockIdx.x + hipThreadIdx.x;
    if(i < N) C[i] = A[i] + B[i];
}

int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
```

```
12      ...

14        float* d_A; hipMalloc(&d_A, size);
          float* d_B; hipMalloc(&d_B, size);
16        float* d_C; hipMalloc(&d_C, size);

18        hipMemcpy(d_A, h_A, size, hipMemcpyHostToDevice);
          hipMemcpy(d_B, h_B, size, hipMemcpyHostToDevice);
20
          int threadsPerBlock = 256;
22        int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
          hipLaunchKernel(VecAdd, blocksPerGrid, threadsPerBlock, 0, 0,
24          d_A, d_B, d_C, N);

26        hipMemcpy(h_C, d_C, size, hipMemcpyDeviceToHost);

28        hipFree(d_A); hipFree(d_B); hipFree(d_C);

30      ...
    }
32
```

Codebeispiel 5.1: Radeon Open Compute

Es gibt nur zwei Unterschiede zu CUDA. Das Präfix cu wird durch hip ersetzt. Diese Funktionen werden in *hip_runtime.h* festgelegt. Außerdem werden die Kernelaufrufe durch die Funktion hipLaunchKernel(<Kernelname>, <dim3: Blöcke>, <dim3 Threads>, <size_t sharedMemorySize >, <hipStream_t: stream>, <ArgumentListe>...); ersetzt. So lassen sich praktisch alle Funktionen der Cuda Runtime API portieren. Nach Setzen der Umgebungsvariable auf oder zur Steuerung des Backends kann das Beispiel mit hcc name.hip.cpp ... übersetzt werden. Eine Library (z.B. cufft.h) würde durch das AMD-Pendant ersetzt werden. Theoretisch lassen sich auch cmake-Dateien mit hipify−cmake automatisch anpassen. Zur Analyse kann rocm−profiler ähnlich zu nvprof verwendet werden.

Da ROC in Teilen auch OpenCL nutzt, wird bei der Installation automatisch eine weitere OpenCL-Plattform hinzugefügt. Dies ist die aktuelle Implementierung von AMD, da das ursprüngliche AMD-APP SDK nicht mehr zur Verfügung steht.

Da die Konzepte von hip lediglich die von CUDA wiederspiegeln, soll hier nicht weiter darauf eingegangen werden. Die grundlegenden Ideen wurden bereits alle behandelt.

HPC Librarys

6. HPC Librarys

Um die Arbeit zu erleichtern, wurden sowohl für OpenCL als auch für CUDA umfangreiche High Performance Bibliotheken implementiert. Beteiligt sind hier die Entwickler von CUDA und OpenCL, eine große Community sowie Forschungsgruppen an Universitäten und privaten Forschungseinrichtungen. Die wichtigsten Themengebiete sind lineare Algebra, Machine Learning, Fast Fourier Transformationen und Monte Carlo Algorithmen, da diese die Grundlage für die meisten Algorithmen in IT und Naturwissenschaften bilden. Die folgenden Kapitel behandeln Einführungen in die Funktionsweise der wichtigsten Bibliotheken für CUDA und OpenCL. Aufgrund des großen Umfangs kann natürlich nur ein kleiner Teil davon gezeigt werden. Es ist daher unumgänglich, in größeren Projekten direkt mit dem Quellmaterial zu arbeiten, also mit den Dokumentationen und Programming Guides der Entwickler.

THRUST

6.1. THRUST: STL für GPUs

THRUST ist die CUDA-Implementierung der Standard Template Library für C++ und ist Teil des CUDA-Toolkits. THRUST orientiert sich an C++11 und beseht aus vier Teilen.

Container

Grundlage für THRUST bilden die Container-Klassen zur Erzeugung dynamischer Datenstrukturen. Diese Klassen befinden sich im Namespace thrust. Die Klasse host_vector entspricht dabei dem C++-vektor. Mit device_vector existiert eine Vektorklasse, die sich ebenfalls über Member-Funktionen steuern lässt, aber zu jedem Zeitpunkt im Speicher des Device liegt.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

...

thrust::host_vector<int> H(4);

H[0] = 14;
H[1] = 20;
H[2] = 38;
H[3] = 46;

std::cout << "H has size " << H.size() << std::endl;

```

```
    H.resize(2);
16
    thrust::device_vector<int> D = H;
18
    D[0] = 99;
20  D[1] = 88;
```

Codebeispiel 6.1: THRUST Vektoren

In diesem Beispiel wird ein Vektor von Ganzzahlen erstellt und dessen Elemnte gesetzt. Die Größe kann über die getter-Funktion size ausgegeben werden. Die Größe kann dynamisch verringert oder vergrößert werden. Danach wird der Hostvektor explizit in einen Devicevektor kopiert. Der Devicevektor lässt sich dabei auch vom Host modifizieren.

Die Idee hinter Devicevektoren ist, Member-Funktionen zu benutzen, die aud diesen Datenstrukturen operieren und dabei in CUDA parallelisiert sind. Devicevektoren werden wie gewöhnliche Vektoren vom Host gesteuert, und können nich direkt im Device Code verwendet werden, da sich im Device Speicher nicht dynamisch verwalten lässt.

Iteratoren

Zur besseren Steuerung der Vektoren bietet THRUST Iteratoren an.

```
    thrust::device_vector<int> D(10, 1);
2   thrust::fill(D.begin(), D.begin() + 7, 9);

4   thrust::host_vector<int> H(D.begin(), D.begin()+5);
    thrust::sequence(H.begin(), H.end());
6
    thrust::copy(H.begin(), H.end(), D.begin());
8
```

Codebeispiel 6.2: THRUST Iteratoren

begin und end sind Iteratoren, die auf Anfang und Ende des belegten Speichers zeigen. So können THRUST Funktionen benutzt werden, um Vektoren zu füllen oder zu Kopieren. Die erste Funktion füllt einen Vektor von Element 0 bis 6 mit 9. Dann wird ein Vektor erstellt

als Kopie der ersten fünf Elemente des vorherigen. sequenze füllt den Vektor mit aufsteigenden Ganzzahlen. Im letzten Schritt wird ein Vektor von Anfang bis Ende An den Beginn eines anderen Vektors kopiert.

Weiter Iteratoren sind

- constant_iterator
- counting_iterator
- transform_iterator
- permutation_iterator
- zip_iterator

und können in Kapitel 4 des Qick Start Guides nachgeschlagen werden. [35]

Funktoren

Funktoren sind spezielle Datentypen, die Funktionen beinhalten, welche bestimmte Operationen ausführen. Diese Operationen sollen Elementweise auf einen Vektor angewendet werden. Folgendes Beispiel definiert einen Funktor für die Saxpy Operation für Host und Device:

```
struct saxpy_functor
{
  const float a;

  saxpy_functor(float _a) : a(_a) {}

  __host__ __device__
  float operator()(const float& x, const float& y) const {return a * x + y;}
};

```

Codebeispiel 6.3: THRUST Funktoren

Nun kann mittels thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A)) diese Operation auf jedes Element von X und Y angewendet und in Y gespeichert werden (siehe

6.1). Es existieren vordefinierte Funktoren wie multiplies oder plus.

Algorithmen

THRUST implementiert schnelle parallele Algorithmen in CUDA für dessen Containerklassen. Neben den genannten Transformationen (siehe 6.1) existiert eine Implementierung eines parallelen Radix-Sortierverfahrens.

```
#include <thrust/sort.h>
#include <thrust/functional.h>

...

const int N = 6;

int A[N] = {1, 4, 2, 8, 5, 7};
thrust::sort(A, A + N);
thrust::sort(A, A + N, thrust::greater<int>());


int    keys[N] = {  1,   4,   2,   8,   5,   7};
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
thrust::sort_by_key(keys, keys + N, values);

```

Codebeispiel 6.4: THRUST Sortieren

sort sortiert einen Vektor numerisch in aufsteigender Reihenfolge (inplace). Dies lässt sich mit einem Funktor kombinieren, um z.B. absteigend zu sortieren. sort_by_keys sortiert ein Array anhand von Eigenschaften eines anderen Arrays, den sogenannten Keys. In diesem Fall wird jedem Buchstaben eine Zahl zugeordnet und die Liste nach diesen Zahlen sortiert. Es existieren ebenfalls stable_sort uns stabel_sort_by_keys, die die relative Ordnung von Elementen mit gleichen Werten, die zur Ordnung klassifizieren, erhalten. Wenn also a und b den gleichen Key erhalten, ist so garantiert, dass a und b nicht vertauscht werden.

Extrem nützlich für GPUs sind die Reduktionen von THRUST.

```
int sum = thrust::reduce(D.begin(), D.end(), (int) 0, thrust::plus<int>());

#include <thrust/count.h>
int result = thrust::count(vec.begin(), vec.end(), 1);
```

Codebeispiel 6.5: THRUST Reduktionen

Erstere Funktion vollzieht eine Summenreduktion, zweitere zählt den Wert eins in einem Vektor. Zum Suchen bestimmter Werte implementiert THUST eine parallele Version einer Binärsuche. Eine Liste aller Reduktionen befindet sich unter http://thrust.github.io/doc/group__reductions.html in der THRUST Dokumentation. [34]

Mittels transform_reduce lassen sich Reduktionen mit selbstgeschriebenen Funktoren kombinieren. Soll eine Reduktion von THRUST auf ein Array angewendet werden, dass von einem Kernel modifiziert wurde oder an eines übergeben werden soll, eignet sich die Klasse device_ptr.

```
size_t N = ...;

int *raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int))

thrust::device_ptr<int> dev_ptr(raw_ptr);
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

thrust::device_ptr<int> dev_ptr = thrust::device_malloc<int>(N);
raw_ptr = thrust::raw_pointer_cast(dev_ptr);
```

Codebeispiel 6.6: THRUST Device Pointer

Im ersten Beispiel wird aus einem gewöhnlichen Devicevektor ein device_ptr Objekt erstellt, im zweiten ein Pointer extrahiert. Auf dieses Objekt kann ein gewöhnlicher THRUST Algorithmus angewendet werden.

Ferner existieren Scan[1]- und Reordering[2] Algorithmen.

[1] http://thrust.github.io/doc/group__prefixsums.html
[2] http://thrust.github.io/doc/group__reordering.html

Random Number Generators

6.2. Random Number Generators

cuRAND

cuRAND ist eine Bibliothek zur Generierung von Pseudo- und Quasi-Zufallszahlen. Sie stellt ein API sowohl für Device als auch für Host zur Verfügung. Zur Benutzung inkludiert man den Header *curand.h* für Host und *curand_kernel.h* für Device. Zusätzlich muss mit *libcurand* gelinkt werden. Um gegen die statische Version zu linken wird folgendes Kommando empfohlen:

```
g++ myCurandApp.c -lcurand_static -lculibos -lcudart_static -lpthread -ldl
```

Pseudo Random

Tabelle 6.1 zeigt die implementierten RNGs. Tabelle 6.3 zeigt alle exisitierenden Verteilungen.

Generator	Beschreibung
CURAND_RNG_PSEUDO_XORWOW	XOR-shift Algorithmus
CURAND_RNG_PSEUDO_MRG32K3A	Combined Multiple Recursive Algorithmus
CURAND_RNG_PSEUDO_MT19937	Mersenne Twister, CPU optimiert verändertes Ordering, nur Host API, SM $\geq$ 3.5
CURAND_RNG_PSEUDO_MTGP32	Mersenne Twister GPU optimierte Parameter
CURAND_RNG_PHILOX4_32_10	non-cryptographic Counter Based Algorithmus

Tabelle 6.1.: Liste der Pseudo RNGs

Zur Benutzung in der Host API muss zunächst ein Buffer erstellt werden. Dann wird ein Generator ausgewählt und mit einem bestimmten Seed initialisiert. Die selben Seeds produzieren die

selben Sequenzen. Anschließend werden die Zufallszahlen anhand einer bestimmten Verteilung erzeugt. Tabelle 6.2 zeigt alle möglichen Verteilungen. Der Seed kann ebenfalls als Zufallszahl gesetzt werden, beispielsweise mit einem Speicherrauschen.

```
    uint n = ...;
2
    float *data;
4   cudaMalloc((void **)&devData, n*sizeof(float));

6   curandGenerator_t gen;
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
8   long long seed = ...;
    curandSetPseudoRandomGeneratorSeed(gen, seed);
10  curandGenerateUniform(gen, devData, n);

12  ...

14  curandDestroyGenerator(gen);
    cudaFree(devData);
16
```

Codebeispiel 6.7: cuRAND: Host API

Die Zufallszahlen können auf den Host zurückkopiert, in einem anderen API verwendet oder in einem eigenen Kernel benutzt werden.

Verteilung	**Beschreibung**
curandGenerate(...)	32bit Zufallszahlen, jedes Bit zufällig
curandGenerateLongLong(...)	64bit Zufallszahlen, jedes Bit zufällig
curandGenerateUniform(...)	Uniforme Zufallszahlen im Bereich $(0, 1]$
curandGenerateNormal(..., float m, float s)	Normalverteilte Zufallszahlen mit Mittelwert und Standardabweichung
curandGenerateLogNormal(..., float mean, float stddv)	Log-Normalverteilte Zufallszahlen
curandGeneratePoisson(..., size_t n, double lambda)	Ganzzahlen der Verteilung (für $\lambda > 0$) $P(k)_\lambda = \lambda^k/k! \cdot e^{-\lambda}$, $k = 0, 1, 2...$
curandGenerateUniformDouble(...)	Doppelte Präzision
curandGenerateNormalDouble(..., float mean, float stddv)	Doppelte Präzision
curandGenerateLogNormalDouble(..., float mean, float stddv)	Doppelte Präzision

Tabelle 6.2.: Liste der Pseudo RNGs

Zur Benutzung des Device APIs muss ein sogenannter State für jeden Thread erstellt werden. Dieser wird in einem eigenen Kernel erstellt. Dazu muss die Funktion curand_init gerufen werden.

Dieser wird ein Seed, eine Sequenznummer und ein Offset übergeben. Der Seed kann auf dem Host zufällig initialisiert werden. Die Sequenznummer hängt vom Threadindex ab und ist damit für jeden Thread verschieden.

```
__global__ void setup_kernel(curandState *state)
{
  uint id = threadIdx.x + blockIdx.x * blockDim.x;
  curand_init(1234, id, 0, &state[id]);
}

...

uint n = ...;
curandState *devStates;
cudaMalloc(&devStates, n*sizeof(curandState));
setup_kernel<<<...,...>>>(devStates);
```

Codebeispiel 6.8: cuRAND: Device API States

Nun kann in einem separaten Kernel anhand dieses States eine Verteilung benutzt werden. So muss pro Programm und Thread die langsame curand_init Funktion nur einmal gerufen werden. Um die Performance weiter zu verbessern, wird jeder State erst in den local Memory kopiert und am Ende des Kernel zurückkopiert. Werden mehrere Zufallszahlen benötigt, wird so der State in den Cache geladen.

```
__global__ void generate_kernel(curandState *state, uint *result)
{
  uint id = threadIdx.x + blockIdx.x * blockDim.x;

  curandState localState = state[id];

  result[id] = curand(&localState);

  state[id] = localState;
}

...

uint *devResults;
```

```
    cudaMalloc(&devResults, n*sizeof(uint));
16  generate_kernel<<<...,...>>>(curandState *state, uint *devResults);
```

Codebeispiel 6.9: cuRAND: Device API Generierung

Benötigt man einen anderen Generator, muss lediglich ein anderer State erzeugt werden:

- curandState (XORWOW)
- curandStatePhilox4_32_10_t
- curandStateMRG32k3a
- curandStateMtgp32_t

In Kapitel 3.1.4 der cuRAND Dokumentation befindet sich eine Liste aller Generatoren. [24]

Quasi Random

Neben den gewöhnlichen Pseudo Zufallszahlen werden des Öfteren sogenannte Quasi Zufallszahlen verwendet. Dies bezeichnet Sequenzen von Zahlen die im mathematischen Sinn eine geringe Diskrepanz aufweisen. Hauptanwendungsgebiet ist Monte-Carlo Integration, da Quasi-Monte-Carlo Integration unter bestimmten Umständen eine höhere Konvergenzgeschwindigkeit aufweist. Die bekannteste und schnellste ist die Sobol Sequenz. Zur Benutzung in der Host API muss lediglich ein entsprechender Generator aus Tabelle 6.3 verwendet werden.

Generator	Beschreibung
CURAND_ RNG_ QUASI_ SOBOL32	32bit Sobol Sequenz
CURAND_ RNG_ QUASI_ SCRAMBLED_ SOBOL32	Scrambled 32bit Sobol Sequenz
CURAND_ RNG_ QUASI_ SOBOL64	64bit Sobol Sequenz
CURAND_ RNG_ QUASI_ SCRAMBLED_ SOBOL64	Scrambled 64bit Sobol Sequenz

Tabelle 6.3.: Liste der Quasi RNGs

Zur Benutzung in der Device API muss ein entsprechender State benutzt werden:

- curandStateSobol32_t

- curandStateScrambledSobol32_t
- curandStateSobol64_t
- curandStateScrambledSobol64_t

In Kapitel 3.2 der cuRAND Dokumentation befindet sich eine Liste der entsprechenden Verteilungen. [24]

Random mit OpenCL

clRNG

http://clmathlibraries.github.io/clRNG/htmldocs/index.html

https://github.com/clMathLibraries/clRNG

```
#define CLRNG_SINGLE_PRECISION
#include <clRNG/mrg31k3p.clh>

__kernel void example(__global clrngMrg31k3pHostStream *streams,
  __global float *out)
{
  int gid = get_global_id(0);

  clrngMrg31k3pStream workItemStream;
  clrngMrg31k3pCopyOverStreamsFromGlobal(1, &workItemStream,
    &streams[gid]);

  out[gid] = clrngMrg31k3pRandomU01(&workItemStream);
}

////////////////////////////////////////////////////////////////////////

cl_int err, streamBufferSize, numWorkItems = ...;

clrngMrg31k3pStream *streams = clrngMrg31k3pCreateStreams(NULL,
  numWorkItems, &streamBufferSize, (clrngStatus *)&err);

cl_mem bufIn = clCreateBuffer(ctx, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
  streamBufferSize, streams, &err);
cl_mem bufOut = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY |
  CL_MEM_HOST_READ_ONLY, numWorkItems * sizeof(cl_float), NULL, &err);
clSetKernelArg(kernel, 0, sizeof(bufIn), &bufIn);
clSetKernelArg(kernel, 1, sizeof(bufOut), &bufOut);

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &numWorkItems,
  NULL, 0, NULL, NULL);

```

Codebeispiel 6.10: clRNG Beispiel

clProbDist

http://umontreal-simul.github.io/clProbDist/htmldocs/index.html

https://github.com/umontreal-simul/clProbDist

```
#include <clProbDist/exponential.clh>

__kernel void example(__global const clprobdistExponential *dist,
  __global double *out)
{
  int gid = get_global_id(0);
  int gsize = get_global_size(0);

  double quantile = (gid + 0.5) / gsize;
  out[gid] = clprobdistExponentialInverseCDFWithObject(dist,
    quantile, (void *)0);
}

////////////////////////////////////////////////////////////////////////

cl_int err, distBufferSize, numWorkItems = ...;
clprobdistExponential *dist;
dist = clprobdistExponentialCreate(1.0, &distBufferSize,
  (clprobdistStatus *)&err);

cl_mem bufIn = clCreateBuffer(ctx, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
  distBufferSize, dist, &err);
cl_mem bufOut = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY |
  CL_MEM_HOST_READ_ONLY, numWorkItems * sizeof(cl_double), NULL, &err);
clSetKernelArg(kernel, 0, sizeof(bufIn), &bufIn);
clSetKernelArg(kernel, 1, sizeof(bufOut), &bufOut);

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &numWorkItems,
  NULL, 0, NULL, NULL);

```

Codebeispiel 6.11: clProbDist Beispiel

clQMC

http://umontreal-simul.github.io/clQMC/htmldocs/index.html

https://github.com/umontreal-simul/clQMC

```
#define CLQMC_SINGLE_PRECISION
#include <clQMC/latticerule.clh>

__kernel void example(__global const clqmcLatticeRule *lat,
  __global float *out)
{
  int gid = get_global_id(0);
  int gsize = get_global_size(0);
  int dim = clqmcLatticeRuleDimension(lat);

  clqmcLatticeRuleStream stream;
  clqmcLatticeRuleCreateOverStream(&stream, lat, gsize, gid, (void*)0);
  for (int j = 0; j < dim; j++)
    out[j * gsize + gid] = clqmcLatticeRuleNextCoordinate(&stream);
}

cl_int err, latBufferSize, numWorkItems = ...;
clqmcLatticeRule *lat;
lat = clqmcLatticeRuleCreate(numWorkItems, 3, (cl_int[]){1, 27, 15},
  &latBufferSize, (clqmcStatus *)&err);

cl_mem bufIn = clCreateBuffer(ctx, CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR,
  latBufferSize, lat, &err);
cl_mem bufOut = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY |
  CL_MEM_HOST_READ_ONLY, numWorkItems*clqmcLatticeRuleDimension(lat)
  *sizeof(cl_float), NULL, &err);
clSetKernelArg(kernel, 0, sizeof(bufIn), &bufIn);
clSetKernelArg(kernel, 1, sizeof(bufOut), &bufOut);

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &numWorkItems,
  NULL, 0, NULL, NULL);

```

Codebeispiel 6.12: clQMC Beispiel

Fast Fourier Transformation

6.3. Fast Fourier Transformation

Theorie

Aus Wikipedia [26]:

Die Fouriertransformation (FT) ist die wohl wichtigste mathematische Transformation im Bereich Daten- und Signalverarbeitung. Sie findet Anwendung in der Audio- und Videobearbeitung, im maschinellen Lernen und in physikalischen und chemischen Simulationen. Da man in Simulationen nur auf diskrete Werte zugreifen kann, wird in solchen üblicherweise die diskrete Form der FT verwendet (DFT):

$$f_m = \sum_{k=0}^{n} x_k \cdot \mathrm{e}^{\left(-\frac{2\pi i}{2n} mk\right)} \qquad m = 0, ..., n-1 \tag{6.1}$$

Wobei f_m das m-te Element der fouriertransformierten eines Vektors (x_0, ..., x_{n-1}) darstellt. Die meisten Algorithmen, die eine DFT implementieren, verwenden die sogeannte Fast Fourier Transformation (FFT): Teilt man den Vektor in gerade x'_k und ungerade Punkte x''_k ein, so folgt für die DFT:

$$f_m = \sum_{k=0}^{n-1} x'_k \cdot \mathrm{e}^{\left(-\frac{2\pi i}{n} mk\right)} \quad + \quad \mathrm{e}^{\left(-\frac{\pi i}{n} m\right)} \sum_{k=0}^{n-1} x''_k \cdot \mathrm{e}^{\left(-\frac{2\pi i}{n} mk\right)} \tag{6.2}$$

$$= \begin{cases} f'_m + \mathrm{e}^{\left(-\frac{\pi i}{n} m\right)} f''_m & m < n \\ f'_{m-n} - \mathrm{e}^{\left(-\frac{\pi i}{n} (m-n)\right)} f''_{m-n} & m \geq n \end{cases} \tag{6.3}$$

Mit der Berechnung von f'_m und f''_m ist sowohl f_m als auch f_{m+n} bestimmt. Der Rechenaufwand hat sich durch diese Zerlegung also praktisch halbiert.

Durch eine Rekursion lässt sich diese Eigenschaft ausnutzen.

1. Das Feld mit den Eingangswerten wird einer Funktion als Parameter übergeben, die es in zwei halb so lange Felder (eins mit den Werten mit geradem und eins mit den Werten mit ungeradem Index) aufteilt.

2. Diese beiden Felder werden nun an neue Instanzen dieser Funktion übergeben.

3. Am Ende gibt jede Funktion die FFT des ihr als Parameter übergebenen Feldes zurück. Diese beiden FFTs werden nun, bevor eine Instanz der Funktion beendet wird, nach der oben abgebildeten Formel zu einer einzigen FFT kombiniert - und das Ergebnis an den Aufrufer zurückgegeben.

Ist n eine zweier-Potenz, so lässt sich der Aufwand effektiv auf $O(n \log n)$ reduzieren. Diese Einschränkung stellt meist kein Hindernis dar, da die Wahl von n in der Praxis oft beliebig ist.

In einer iterativen Variante lässt sich dieser Algorithmus auf einer GPU parallelisieren: In jedem Schritt werden die geraden und ungeraden Punkte in zwei Hälften geteilt. Man erhält ein sortiertes Array, in dem von jeweils zwei Elementen die DFT gebildet werden kann. Im nächsten Schritt werden diese Werte zur DFT von vier Werten kombiniert. Dies führt man so lange durch, bis man auf der obersten Ebene angelangt ist.

Die inverse Transformation unterscheidet sich nur durch Vorzeichen und eine Normierungskonstante.

Dieses Vorgehen ähnelt stark einer Reduktion. Es ist daher nicht überraschend, dass auch hier eine CUDA Library als Teil des Toolkits zur Verfügung steht.

cuFFT

Bei der cuFFT Library handelt es sich um die state-of-the-art Implementierung einer GPU parallelisierten FFT. Sie wurde im Wesentlichen von der *Fastest Fourier Transformation in the West* (FFTW) inspiriert, unterscheidet sich aber in einem wichtigen Punkt (siehe 6.3). Zur Benutzung muss der Header *cufft.h* (bzw. *cufftXt.h* für Xt Funktionalität) inkludiert und mit *libcufft* gelinkt werden.

cuFFT unterstützt FFTs in bis zu drei Dimensionen von Komplex nach Komplex (CUFFT_C2C), Real nach Komplex (CUFFT_ R2C) und Komplex nach Real (CUFFT_ C2R). R und C bezeichnen dabei einfache Präzision. Für die doppelte verwendet man D und Z.

Zunächst müssen Arrays eines speziellen cuFFT Datentyps (cufftComplex. cufftReal, cufftDouble, cufftDoubleComplex) erstellt und wie gewohnt in den global Memory kopiert werden:

```
int dim = {Nx, Ny, Nz};
cufftComplex *data;
cudaMalloc(&data, Nx*Ny*Nz * sizeof(cufftComplex));

```

Codebeispiel 6.13: cuFFT: komplexer Datentyp

Ein Wert lässt sich dann wie in einem multidimensionalen Array von zweikomponentigen Datenstrukturen setzen, z.B. `data[x][y][z].x = 5.0f; data[x][y][z].y = 3.0f;` für $5 + i \cdot 3$.

Im nächsten Schritt muss ein Handle für das Programm, ein sogenannter Plan erstellt werden. Dies ist notwendig, da abhängig von der Beschaffenheit der Daten vorab bereits verschiedene Berechnungen durchgeführt werden müssen, z.B. die Größe der Blöcke oder welcher Algorithmus benutzt werden soll.

```
cufftHandle plan;
////////in einer Dimension ////////
cufftPlan1d(&plan, Nx, CUFFT_C2C, 1);
////////in zwei Dimensionen////////
cufftPlan2d(&plan, Nx, Ny, CUFFT_C2C, 1);



////////in drei Dimensionen////////
cufftPlan3d(&plan, Nx, Ny, Nz, CUFFT_C2C, 1);

```

Codebeispiel 6.14: cuFFT: Pläne

Der letzte Wert bezeichnet die Batch Größe (später mehr).

Dieser Plan muss nun ausgeführt und evtl. das Ergebnis zur Ausgabe zurückkopiert werden.

```
    cufftExecC2C(plan, data, data, CUFFT_FORWARD);
2   cufftDestroy(plan);
    cudaFree(data);
4
```

Codebeispiel 6.15: cuFFT: Ausführen

Input und Output Array sind hier identisch, es handelt sich also um eine in-place Transformation. Das letzte Keyword bezeichnet die Art der Transformation. CUFFT_ INVERSE bezeichnet die inverse Transformation. Da Input und Output Array nicht zwingend vom selben Datentyp sein müssen, können ihre Größen abweichen. Tabelle 6.4 zeigt einen Vergleich.

Dimension	FFT	Inputgröße	Outputgröße
1D	C2C	N_x cufftComplex	N_x cufftComplex
1D	C2R	$\left\lfloor \frac{N_x}{2} \right\rfloor + 1$ cufftComplex	N_x cufftReal
1D	R2C	N_x cufftReal	$\left\lfloor \frac{N_x}{2} \right\rfloor + 1$ cufftComplex
2D	C2C	$N_x \cdot N_y$ cufftComplex	$N_x \cdot N_y$ cufftComplex
2D	C2R	$N_x \left\lfloor \frac{N_y}{2} \right\rfloor + 1$ cufftComplex	$N_x \cdot N_y$ cufftReal
2D	R2C	$N_x \cdot N_y$ cufftReal	$N_x \left\lfloor \frac{N_y}{2} \right\rfloor + 1$ cufftComplex
3D	C2C	$N_x \cdot N_y \cdot N_z$ cufftComplex	$N_x \cdot N_y \cdot N_z$ cufftComplex
3D	C2R	$N_x \cdot N_y \left\lfloor \frac{N_z}{2} \right\rfloor + 1$ cufftComplex	$N_x \cdot N_y \cdot N_z$ cufftReal
3D	R2C	$N_x \cdot N_y \cdot N_z$ cufftReal	$N_x \cdot N_y \left\lfloor \frac{N_z}{2} \right\rfloor + 1$ cufftComplex

Tabelle 6.4.: Größenvergleich von Input und Output Arrays

Nvidia stellt auch hier eine Dokumentation im gewohnten Format zur Verfügung. [10]

cuFFT v.s. FFTW

Die Fastest Fourier Transformation in the West (FFTW) hat sich wegen ihrer freien Zugänglichkeit und ihrer effizienten Parallelisierung in OpenMP und MPI für Prozessoren als de-facto Standard durchgesetzt. Sie unterscheidet sich in einem wesentlichen Punkt gegenüber cuFFT: FFTW stellt viele Pläne zur Verfügung und führt sie auf eine Art aus, cuFFT stellt nur drei

Pläne zur Verfügung und benutzt mehrere Methoden zur Ausführung [14]. Tabelle 6.5 zeigt einige Unterschiede.

FFTW	cuFFT
fftw_plan_dft_1d(), fftw_plan_dft_r2c_1d(), fftw_plan_dft_c2r_1d()	cufftPlan1d()
fftw_plan_dft_2d(), fftw_plan_dft_r2c_2d(), fftw_plan_dft_c2r_2d()	cufftPlan2d()
fftw_plan_dft_3d(), fftw_plan_dft_r2c_3d(), fftw_plan_dft_c2r_3d()	cufftPlan3d()
fftw_plan_dft(), fftw_plan_dft_r2c(), fftw_plan_dft_c2r()	cufftPlanMany()
fftw_plan_many_dft(), fftw_plan_many_dft_r2c(), fftw_plan_many_dft_c2r()	cufftPlanMany()
fftw_execute()	cufftExecC2C() cufftExecZ2Z() cufftExecR2C() cufftExecD2Z() cufftExecC2R() cufftExecZ2D()
fftw_destroy_plan()	cufftDestroy()

Tabelle 6.5.: Unterschiede zwischen FFTW und cuFFT

Eine eindimensionale Transformation würde mit FFTW in der OpenMP-parallelen Variante so aussehen:

```
    fftw_complex *in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*N);
2   //Komplexen Vektor belegen...
    fftw_init_threads();
4   fftw_plan_with_nthreads(omp_get_max_threads());
    fftw_plan p = fftw_plan_dft_1d(N, in, in, FFTW_FORWARD, FFTW_ESTIMATE);
6   fftw_execute(p);

8   fftw_destroy_plan(p);
    fftw_free(in);
10  fftw_cleanup_threads();
```

Codebeispiel 6.16: FFTW Beispiel

Abbildung 6.1 zeigt einen Laufzeitvergleich.

Mehr Informationen lassen sich in der Dokumentation nachlesen. [13]

cuFFTW

Um eine Umstellung zu erleichtern, wurde ein API programmiert, das manche FFTW Funktionen auf cuFFT abbildet. Dazu muss lediglich der Header gegen *cufftw.h* ausgetauscht und mit *libcufftw* gelinkt werden. Kapitel 7 in der Dokumentation zeigt alle implementierten Funktionen. [10]

clFFT

Es existiert eine Open-Source Implementierung der FFT in OpenCL von AMD. Sie wurde zwar für AMD GPUs optimiert lässt sich aber prinzipiell auf allen benutzen. Diese steht unter `https://github.com/clMathLibraries/clFFT` zur Verfügung und muss explizit nach Anleitung kompiliert werden. Zur Benutzung muss der Header *clfft.h* inkludiert und mit *libclFFT* (und wie gewohnt mit *libOpenCL*) gelinkt werden. Eine eindimensionale Transformation würde mit clFFT so aussehen:

```
cl_int err, N = ...;
float *X = (float*)malloc(N*2 * sizeof(*X));
//Komplexen Vektor belegen...

cl_context ctx = ...;
cl_command_queue queue = ...;

clfftSetupData fftSetup;
clfftInitSetupData(&fftSetup);
clfftSetup(&fftSetup);

cl_mem bufX = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
  N*2 * sizeof(*X), NULL, &err);
clEnqueueWriteBuffer(queue, bufX, CL_TRUE, 0,
  N*2 * sizeof(*X), X, 0, NULL, NULL);

clfftDim dim = CLFFT_1D;
size_t clLengths[1] = {N};
clfftPlanHandle planHandle;
clfftCreateDefaultPlan(&planHandle, ctx, dim, clLengths);

clfftSetPlanPrecision(planHandle, CLFFT_SINGLE);
clfftSetLayout(planHandle, CLFFT_COMPLEX_INTERLEAVED,
                           CLFFT_COMPLEX_INTERLEAVED);
clfftSetResultLocation(planHandle, CLFFT_INPLACE);

clfftBakePlan(planHandle, 1, &queue, NULL, NULL);
clfftEnqueueTransform(planHandle, CLFFT_FORWARD, 1,
  &queue, 0, NULL, NULL, &bufX, NULL, NULL);

.........

clfftDestroyPlan(&planHandle);
clfftTeardown();
```

Codebeispiel 6.17: clFFT Beispiel

Abbildung 6.1 zeigt einen Laufzeitvergleich.

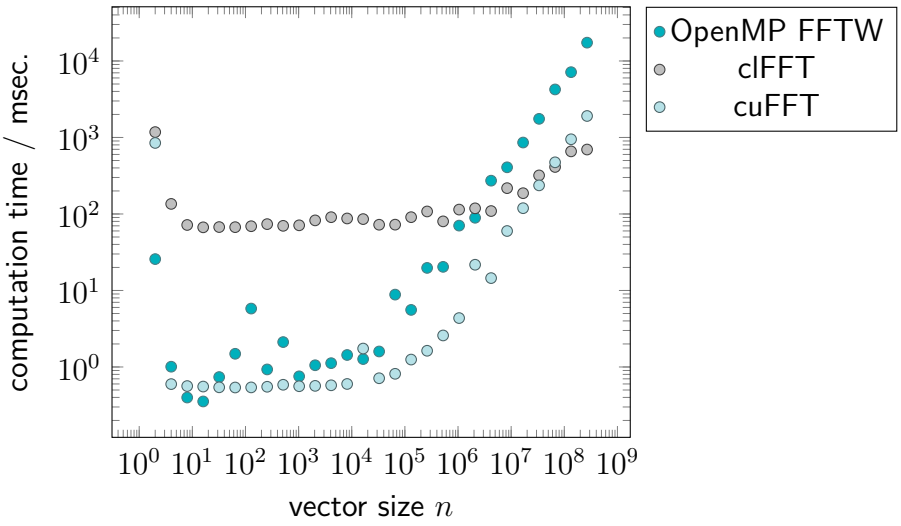

Abbildung 6.1.: Laufzeitvergleich von *Complex-to-Complex* FFTs desselben n-dimensionalen Vektors (log-log). Zum Einsatz kam eine *Nvidia GTX 1060* sowie ein Intel i7-8700K 6×4.8GHz.

Mehr Informationen zum API lassen sich in der Dokumentation nachlesen. [5]

Lineare Algebra

6.4. Lineare Algebra

cuBLAS und clBLAS

Um für Programmierer das Implementieren von Algorithmen in der linearen Algebra zu erleichtern, wurden in den 70er Jahren in Fortran verschiedenen Matrix- und Vektoroperationen hardwarenah implementiert. Diese Bibliothek ist als *Basic Linear Algebra Subprograms* (BLAS) bekannt. Es exisiteren Wrapper und Portierungen für verschiedene Sprachen.

BLAS besteht aus drei Teilen:

- **Level 1: Vektor-Vektor Operationen**
  Normen, Skalarprodukte, Dotprodukte, Kreuzprodukte, ...
- **Level 2: Matrix-Vektor Operationen**
  Multiplikationen von Vektoren und Matrizen unter Nebenbedingungen (z.B. symmetrische Matrix, ...)
- **Level 3: Matrix-Matrix Operationen**
  Matrixnormen, Matrixadditionen, Matrixmultiplikationen unter Nebenbedingungen, ...

Es existiert eine Implementierung von Nvidia in CUDA namens cuBLAS. In dieser Bibliothek, die Teil des CUDA Toolkits ist, werden diese Operationen auf der GPU parallelisiert. Zur Benutzung muss der Header *cublas_v2.h* (in älteren Versionen *cublas.h*) inkludiert und mit *libcublas* gelinkt werden.

Die Handhabung soll möglichst einfach gehalten werden. Der Anwender implementiert Vektoren als gewöhnliche Arrays und linearisierte Matrizen ebenfalls als Array. Für komplexe Zahlen existiert das Datenformat cuComplex (cuDoubleComplex), welches analog zu cufftComplex (cufftDoubleComplex) funktioniert. In cuBLAS geht man von column-major Format und 1-based indexing aus. Dabei wird eine Matrix spaltenweise nacheinander in einem Array abgelegt. Dann ist für ein Element einer Matrix $A(i, j)$ (wobei i und j mit 1 beginnen) der Index im Array A[(j−1)∗m + i−1], wobei m die Anzahl an Reihen einer Matrix ist.

```
#define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))

uint M = ...;
uint N = ...;

float* a = (float *)malloc (M * N * sizeof(float));
for (uint j = 1; j <= N; j++)
{
  for (uint i = 1; i <= M; i++) a[IDX2F(i,j,M)] = ...;
}
float *b = ...; float *c = ...;

float* devPtrA;
cudaMalloc(&devPtrA, M*N*sizeof(float));

float* devPtrA;
float* devPtrB;
float* devPtrC;
cublasSetMatrix(M, N, sizeof(float), a, M, devPtrA, M);
...
```

Codebeispiel 6.18: cuBLAS: Matrix setzen

Sobald die Objekte gesetzt wurden, wird ähnlich wie in cuFFT ein Handle erstellt. Dann können auf diesen Objekten bestimmte Operationen ausgeführt werden, deren Namen und Handhabung

in Kapitel 2.4 der cuBLAS Dokumentation [37] nachgeschlagen werden können. In den meisten Fällen existieren Funktionen in mehreren Versionen für verschiedenen Präzisionen. Nach Ausführung kann das Objekt zur Ausgabe in den CPU Speicher zurückkopiert werden. Typischerweise implementieren diese Funktionen eine Abfolge von mehreren Operationen, die auf eine spezielle Operation angepasst werden muss.

Man betrachte die Berechnung einer Matrixmultiplikation $C = A \cdot B$ einer $m \times n$ Matrix A und einer $n \times m$ Matrix B im allgemeinen Fall für einfache Präzision. Dafür existiert die Funktion cublasSgemm. Diese implementiert die Abbildung $C \leftarrow \alpha \cdot \mathbf{op}(A)\mathbf{op}(B) + \beta \cdot C$, wobei op() für transponieren oder konjugieren stehen kann und α bzw. β beliebige Skalare sind. Für op() existiert der Datentyp cublasOperation_t und kann entweder CUBLAS_OP_N („normal“), CUBLAS_OP_T („transponiert“) oder CUBLAS_OP_C („komplex konjugiert“) sein. In diesem Beispiel muss also die Funktion wie folgt gerufen werden:

```
    cublasHandle_t handle;
2   cublasCreate(&handle);

4   cublasSgemm(handle,
      CUBLAS_OP_N , CUBLAS_OP_N, M, M, N,
6     1.0, devPtrA, M, devPtrB, N,
      0.0, devPtrC, M);
8
    cublasGetMatrix(M, M, sizeof(float), devPtrC, M, c, M);
10
    cudaFree(devPtrA); cudaFree(devPtrB); cudaFree(devPtrC);
12  free(a); free(b); free(c);

14  cublasDestroy(handle);
```

Codebeispiel 6.19: cuBLAS: Funktionsaufruf

Zunächst muss eine $m \times m$ Matrix C mit null vorbelegt werden. Die Größen von A und B müssen richtig angegeben und mit der Operation CUBLAS_OP_N versehen werden. Dazu wird nach dem Devicepointer der Matrix die Leading Dimension angegeben. Die zweite Dimension ergibt sich aus der Gesamtgröße des Arrays. Zudem müssen Länge und Breite noch einmal explizit angegeben werden, um evtl. nur eine Submatrix zu multiplizieren. Skalar α wird auf eins, β auf null gesetzt.

In älteren Versionen sind Wrapper für die Allozierungsfunktionen vorhanden. Diese gelten als veraltet und sollten nicht mehr verwendet werden.

Es existiert eine von AMD entwickelte Portierung clBLAS für OpenCL. Wie gewohnt ist diese Bibliothek quelloffen und kann unter `https://github.com/clMathLibraries/clBLAS` heruntergeladen werden. Auch diese Library wurde für AMD GPUs optimiert, lässt sich aber auch auf anderen Geräten verwenden. Eine Dokumentation steht ebenfalls zur Verfügung [4]. Nach Installation muss der Header *clblas.h* inkludiert und mit *libclBLAS* (und wie gewohnt mit *libOpenCL*) gelinkt werden.

Zur Benutzung muss eine Initialisierungsfunktion gerufen werden. Matrizen werden ebenfalls im column-major Format in einen Buffer geschrieben. Da sich clBLAS ebenfalls an BLAS orientiert, unterscheiden sich die Namen der Funktionen gegenüber cuBLAS nur im Präfix „cl“. Nach Ausführen dieser Funktion muss synchronisiert und das Ergebnis zurückkopiert werden.

```
    cl_int err;
2   cl_uint M = ...; cl_uint N = ...;

4   cl_float *A = ...; cl_float *B = ...; cl_float *C = ...;

6   clblasSetup();

8   cl_mem bufA = clCreateBuffer(ctx, CL_MEM_READ_ONLY,
      M*N * sizeof(cl_float), NULL, &err);
10  cl_mem bufB = ...; cl_mem bufC = ...;

12  clEnqueueWriteBuffer(queue, bufA, CL_TRUE, 0,
      M*N * sizeof(cl_float), A, 0, NULL, NULL);
14
    cl_event event;
16  clblasSgemm(clblasColumnMajor, clblasNoTrans, clblasNoTrans, M, M, N,
      1.0, bufA, 0, M, bufB, 0, N,
18    0.0, bufC, 0, M,
      1, &queue, 0, NULL, &event);
20  clWaitForEvents(1, &event);

22  clEnqueueReadBuffer(queue, bufC, CL_TRUE, 0,
      M*M * sizeof(cl_float), C, 0, NULL, NULL);
24
```

```
    clReleaseMemObject(bufC);clReleaseMemObject(bufB);clReleaseMemObject(bufA);
26  clblasTeardown();
```

Codebeispiel 6.20: clBLAS Beispiel

Einziger Unterschied zu cuBLAS ist die Angabe von OpenCl-spezifischen Eigenheiten, wie z.B. die Angabe eines Offsets oder der Events.

cuSPARSE und clSPARSE

In vielen Fällen der Praxis handelt es sich bei Matrizen um sogenannte dünn besetzte Matrizen (engl. *sparse matrices*). Das Prinzip eines sparse-Formates beruht darauf, die wesentliche Anzahl von Nullen in der Matrix nicht zu speichern. Man speichert lediglich die von null verschiedenen Werte und deren Indizes in der Matrix. Bei einer Bandmatrix z.B. steigt die Zahl dieser Elemente linear mit n, die Zahl der Nullen aber quadratisch. Daher ist in den meisten Fällen ein solches Format bereits wegen dem geringeren Speicherbedarf sinnvoll.

Eine Multiplikation mit Null ergibt immer Null und muss nicht explizit ausgeführt werden. Um die Performance zu steigern, müssen folglich Algorithmen verwendet werden, die dieses Format explizit unterstützen. Für CUDA existiert die Bibliothek cuSPARSE. Das Inkludieren und Linken erfolgt wie gewohnt (*cusparse.h*, *libcusparse*).

Folgendes Beispiel zeigt die Erstellung einer Matrix im *Coordinated Format* (COO). Wie bei cuBLAS existiert auch hier kein expliziter Datentyp. Für die Matrixwerte und die Indizes werden eigene Arrays erstellt und manuell belegt.

```
    int n   = ...; int nnz = ...;
2   int *cooRowIndexHostPtr =    (int*)malloc(nnz*sizeof(int));
    int *cooColIndexHostPtr =    (int*)malloc(nnz*sizeof(int));
4   double *cooValHostPtr   = (double*)malloc(nnz*sizeof(double));

6   //Beispiel (COO Format):
    cooRowIndexHostPtr[0]=0; cooColIndexHostPtr[0]=0; cooValHostPtr[0]=1.0;
8   cooRowIndexHostPtr[1]=0; cooColIndexHostPtr[1]=2; cooValHostPtr[1]=2.0;
    ...
10
```

Codebeispiel 6.21: cuSPARSE: Matrix erstellen

Bei diesem Format werden die Werte und deren Indizes in der Matrix nacheinander, ohne Ordnung und zeilenweise abgelegt. Es existieren komplexere Formate, die für bestimmte Algorithmen notwendig sind. Für die Operation aus dem letzten Abschnitt wird beispielsweise das *Block Compressed Sparse Row Format* (BSR) Format benötigt. Die genauen Anforderungen der Formate können in der Dokumentation nachgelesen werden. [11] Für Vektoren ist die Angabe eines zweiten Index natürlich überflüssig. Dichte Vektoren werden als gewöhnliche Arrays gespeichert.

```
    int nnz_vector = ;
2   int    *xIndHostPtr =    (int*)malloc(nnz_vector*sizeof(int));
    double *xValHostPtr = (double*)malloc(nnz_vector*sizeof(double));
4
    double *yHostPtr    = (double*)malloc(2*n       *sizeof(double));
6
    double *zHostPtr    = (double*)malloc(2*(n+1)   *sizeof(double));
8
```

Codebeispiel 6.22: cuSPARSE: Vector erstellen

Sämtliche Arrays müssen wie üblich auf der Grafikkarte alloziert und vom Host kopiert werden. Danach wird ein Handle und ein Matrix Deskriptor erstellt, der eine Datenstruktur für das Device aufbaut. Die Angabe des Matrixtyps und der Art der Indizierung ist dabei erforderlich. Für Ersteres existieren folgende Möglichkeiten:

- CUSPARSE_MATRIX_TYPE_GENERAL
- CUSPARSE_MATRIX_TYPE_SYMMETRIC
- CUSPARSE_MATRIX_TYPE_HERMITIAN
- CUSPARSE_MATRIX_TYPE_TRIANGULAR

Dadurch lassen sich beim Speichern redundante Informationen vermeiden.

Die Library implementiert zudem einige Konvertierungsfunktionen, z.B. vom COO ins CSR Format. Wichtig ist die Angabe von nnz, die Anzahl der von null verschiedenen Elemente.

```
//Malloc und Memcopy
...

cusparseHandle_t handle;
cusparseCreate(&handle);

cusparseMatDescr_t descr;
cusparseCreateMatDescr(&descr);
cusparseSetMatType(descr,CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase(descr,CUSPARSE_INDEX_BASE_ZERO);

int *csrRowPtr;
cudaMalloc(&csrRowPtr, (n+1)*sizeof(int));
cusparseXcoo2csr(handle, cooRowIndex, nnz, n,
                 csrRowPtr, CUSPARSE_INDEX_BASE_ZERO);

```

Codebeispiel 6.23: cuSPARSE: Initialisierung und Konvertierung

Auch hier existieren drei Level an Operationen.

- **Level 1: sparseVektor - denseVektor Operationen**
- **Level 2: sparseMatrix - denseVektor Operationen**
- **Level 3: sparseMatrix - mehrere Vektoren gespeichert als denseMatrix**

Bei Letzterem werden die Vektoren nacheinander im selben Array abgespeichert. Im Folgenden wird jeweils ein Beispiel gezeigt.

```
//Level 1
cusparseDsctr(handle, nnz_vector, xVal, xInd,
              &y[n], CUSPARSE_INDEX_BASE_ZERO);


//Level 2
```

```
    cusparseDcsrmv(handle,CUSPARSE_OPERATION_NON_TRANSPOSE, n, n, nnz,
8                  &number, descr, cooVal, csrRowPtr, cooColIndex,
                   &y[0], &number, &y[n]);
10
    //Level 3
12  double *z; cudaMalloc(&z,    2*(n+1)*sizeof(double));
    cudaMemset(z, 0, 2*(n+1)*sizeof(double));
14  cusparseDcsrmm(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, n, 2, n,
                   nnz, &number, descr, cooVal, csrRowPtr, cooColIndex,
16                 y, n, &number, z, n+1);

18  cusparseDestroyMatDescr(descr);
    cusparseDestroy(handle);
20
```

Codebeispiel 6.24: cuSPARSE: Beispiele und Cleanup

Handle und Deskriptor müssen explizit zerstört werden. Wie gewohnt müssen Arrays auf Host und Device freigegeben werden.

cuSOLVER

Es exisitiert eine Reihe von Bibliotheken, die aufbauend auf BLAS die wichtigsten Algorithmen der linearen Algebra implementieren, z.B. QR-Zerlegung oder Eigenwertberechnung. Die bekannteste ist das *Linear Algebra Package* (LAPACK), das in Fortran77 geschrieben wurde (umgeschrieben auf Fortran90) und für die meisten Programmiersprachen portiert wurde. Es existieren hierfür auch APIs, z.B. Armadillo oder JLapack. Um für dicht und dünn besetzte Matrizen schnelle Methoden für Berechnungen in der linearen Algebra bereit zu stellen, existiert eine Implementierung der wichtigsten Algorithmen in der Bibliothek cuSOLVER, ein Teil des CUDA Toolkits. Linken und Inkludieren erfolgt wie gewohnt (*cusolver.h*, *libcusolver*). Diese Bibliothek orientiert sich an LAPACK und besteht aus drei Teilen:

- **cuSolverDN: Dense LAPACK**

  Methoden zur Lösung eines regulären dicht besetzten Gleichungssystems $Ax = b$ (QR- und LU-Zerlegung mit Pivotisierung, Cholesky Zerlegung für symmetrisch/hermitesche Matrizen, Bunch-Kaufmann-Zerlegung für symmetrisch indefinite Matrizen, Singulär-

wertzerlegung).

- **cuSolverSP: Sparse LAPACK**

  Methoden zur Lösung eines dünn besetzten Gleichungssystems $Ax = b$, im überbestimmten Fall durch Regression $x = \text{argmin}||Az - b||$, im unterbestimmten Fall durch Lösen von $A^T x = b$. Für symmetriche Matrizen muss eine volle Matrix angegeben werden (Ausnahme: Bei Cholesky-Zerlegung reicht die untere linke Dreiecksmatrix.) Zusätzlich existiert eine Eigenwertzerlegung auf Basis der shift-inverse-power Methode.

- **cuSolverRF: Refactorization**

  Methoden zur schnellen Lösung einer Menge von Gleichungssystemen $A_i x_i = b_i$. Bleibt das sparsity-pattern[3] der Matrizen A_i, sowie das Neuordnen zur Pivotisierung und zum Auffüllen während der LU-Zerlegung gleich, ist dieser Teil der Bibliothek anwendbar. Zuerst wird für $i = 1$ eine volle LU-Zerlegung durchgeführt. Für die folgenden kann ein LU-Refaktorisierungsalgorithmus verwendet werden.

Jede Operation ist grundsätzlich für die Datentypen float , double, cuComplex, und cuDoubleComplex implementiert. Für die Funktionen gelten bestimmte Namenskonventionen. Für Dense:

cusolverDn<t><operation>

Der Datentyp wird mit <t> angegeben, S (single), D (double), C (complex single), Z (complex double) und X (generic).

Für Sparse:

cusolverSp[Host]<t>[<matrix data format>]<operation>[<output matrix data format>]<based on>

Die Angabe [Host] steht für die CPU-Implementierung. [<matrix data format>] steht für die Angabe des sparse-matrix Formats (z.B. csr). [<output matrix data format>] ist entweder m oder v für Matrix oder Vektor. <based on> beschreibt den Algorithmus (z.B. qr).

Im Folgenden soll die Lösung eines regulären linearen Gleichungssystems $A_{m \times m} x = B$ mittels LU-Zerlegung ermittelt werden.

[3] Position der von null verschiedenen Elemente

```
const int m = ...; const int lda = m; const int ldb = m;
float A[lda*m] = {...};
float B[m] = {...};

cusolverDnHandle_t cusolverH; cusolverDnCreate(&cusolverH);
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
cusolverDnSetStream(cusolverH, stream);

float *d_A; cudaMalloc(&d_A, sizeof(float)*lda*m);
float *d_B; cudaMalloc(&d_B, sizeof(float)*m);

cudaMemcpy(d_A, A, lda*m*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B,     m*sizeof(float), cudaMemcpyHostToDevice);

int lwork; float *d_work;
cusolverDnDgetrf_bufferSize(cusolverH, m, m, d_A, lda, &lwork);
cudaMalloc(&d_work, sizeof(float)*lwork);
```

Codebeispiel 6.25: cuSOLVER: Buffer

Für die Nutzung der Funktionen muss ein Handle erzeugt und einem Stream hinzugefügt werden. Belegen der Matrizen erfolgt wie gewohnt linearisiert. Allerdings muss danach mit cusolverDnDgetrf_bufferSize die Größe eines Buffers, der für die Zerlegung notwendig ist, ermittelt werden. Mit dieser Angabe wird dann der Buffer explizit angelegt.

```
int Ipiv[m];
int info;
int *d_Ipiv; cudaMalloc(&d_Ipiv, m*sizeof(int));
int *d_info; cudaMalloc(&d_info,   sizeof(int));

const int pivot_on = 1;
if(pivot_on)
  cusolverDnDgetrf(cusolverH, m, m, d_A, lda,
  d_work, d_Ipiv, d_info);
else
  cusolverDnDgetrf(cusolverH, m, m, d_A, lda,
  d_work, NULL,   d_info);
```

```
14    cudaDeviceSynchronize();

16    float LU[lda*m];
      if(pivot_on)
18      cudaMemcpy(Ipiv, d_Ipiv, m*sizeof(int), cudaMemcpyDeviceToHost);
      cudaMemcpy(LU, d_A, lda*m*sizeof(float), cudaMemcpyDeviceToHost);
20    cudaMemcpy(&info, d_info, sizeof(int), cudaMemcpyDeviceToHost);

22    if(0 > info){
        printf("%d-th parameter is wrong \n", -info); exit(1);
24    }
```

Codebeispiel 6.26: cuSOLVER: LU-Zerlegung

Nun kann explizit eine LU-Zerlegung $A = LU$ mittels cusolverDnDgetrf durchgeführt werden. Dies funktioniert optional mit einer Pivotisierung $PA = LU$ (Permutationsmatrix P), bei der zusätzlich das Pivot Element jeder Zeile gespeichert wird. Bei dem speziellen Info „Array“ d_info handelt es sich um einen einzigen globalen Integer-Wert, der mit negativem Vorzeichen angibt, welcher Parameter bei der Berechnung fehlerhaft war. Bei einer fehlerfreien Berechnung ist dieser Wert mit null belegt. Die Ausgabe der fertige Zerlegung wird als $(L-1)+U$ in einer einzigen Matrix gespeichert.

```
      if(pivot_on)
2       cusolverDnDgetrs(cusolverH, CUBLAS_OP_N, m, 1, d_A, lda,
        d_Ipiv, d_B, ldb, d_info);
4     else
        cusolverDnDgetrs(cusolverH, CUBLAS_OP_N, m, 1, d_A, lda,
6       NULL,    d_B, ldb, d_info);

8     cudaDeviceSynchronize();

10    float X[m];
      cudaMemcpy(X, d_B, m*sizeof(float), cudaMemcpyDeviceToHost);
12
      cudaFree(d_A); cudaFree(d_B); cudaFree(d_Ipiv);
14    cudaFree(d_info); cudaFree(d_work);

16    cusolverDnDestroy(cusolverH);
```

```
    cudaStreamDestroy(stream);
18
```

Codebeispiel 6.27: cuSOLVER: Gleichungssystem lösen

Der Aufruf führt cusolverDnDgetrs (wahlweise mit Pivotisierung) eine Vor- und Rückwärtssubstitution aus und berechnet so die Lösung des Gleichungssystems ($Ly = B$, $Ux = y$). Das Ergebnis steht in d_B bereit.

Für eine Operation aus Level 2 muss eine dünne Matrix im richtigen Format analog zu cuSPARSE erstellt werden. Dann wir ein entsprechender Handle mit cusolverSpHandle_t erstellt und die entsprechende Funktion ausgeführt. Da dies recht aufwändig ist und keinen großen Mehrwert bietet, soll hier darauf verzichtet werden.

MAGMA

Die zunehmende Notwendigkeit, Algorithmen auf heterogenen Systemen zu parallelisieren, motivierte die Entwicklung der *Matrix Algebra on GPU and Multicore Architectures* (MAGMA). MAGMA orientiert sich in der Funktionsweise an BLAS bzw. LAPACK und soll effiziente Algorithmen implementieren, die die Hardware eines heterogenen Systems (gleichzeitig Vielkernprozessor und verschiedene GPUs) vollständig ausreizen. Bei Installation wird Existenz und Version von CUDA und OpenMP abgefragt. Dann wird die Bibliothek für alle relevanten Architekturen und Konfigurationen kompiliert. Linken erfolgt dann gegen *libmagma* oder *libmagma_sparse* und gegen die entsprechenden Abhängigkeiten (*liblapack*, *libcuBLAS*, ...). Inkludiert wird *magma_v2.h* für die BLAS- und *magma_lapack.h* für die LAPACK-Portierung.

Es existieren zwei Interfaces. Falls keine Grafikkarte zur Verfügung steht, kann das CPU-Interface benutzt werden. In diesem Fall wird ein OpenMP parallelisierter Algorithmus verwendet. Andernfalls kann mittels GPU-Interface ein mit CUDA beschleunigter Hybridalgorithmus verwendet werden. Sollte trotz vorhandener GPU (und GPU-Installation) das CPU-Interface benutzt werden, wird dennoch ein Hybridalgorithmus verwendet.

Im Folgenden soll die LU-Zerlegung in cuSOLVER mit MAGMA durchgeführt werden.

```
    magma_init();
```

```
    magma_queue_t queue;
    magma_int_t  dev = 0;
    magma_queue_create(dev, &queue);

    magma_int_t m = ...; magma_int_t n = 1;


    float *a; magma_smalloc_pinned(&a, m*m);
    float *b; magma_smalloc_pinned(&b, m);
    float *x; magma_smalloc_pinned(&x, m);
    magma_int_t *piv = (magma_int_t *)malloc(m*sizeof(magma_int_t));

    //Belegen...

    const float alpha = 1.0f;
    const float beta  = 0.0f;
    blasf77_sgemm("N", "N", &m, &n, &n, &alpha, a, &m, x, &m, &beta, b, &m);

    magma_int_t info;
    magma_sgesv(m, n, a, m, piv, b, m, &info);

    magma_free_pinned(a); magma_free_pinned(b); magma_free_pinned(x); free(piv);

    magma_queue_destroy(queue);
    magma_finalize();
```

Codebeispiel 6.28: MAGMA: CPU-Interface

Zunächst wird MAGMA initialisiert. Danach wird ein Device ausgewählt und einer Command Queue hinzugefügt. Bei der Verwendung des CPU Interfaces für einen GPU Algorithmus wird managed Memory verwendet. Daher muss für die Arrays pinned Memory (page-locked) alloziert werden. Danach erfolgt ein Aufruf einer Matrixmultiplikation analog zu cuBLAS (blasf77_sgemm), um die Lösung später überprüfen zu können. Die Zerlegung selbst erfolgt dann analog zu cuSOLVER mittels magma_sgesv.

```
    magma_init();
    magma_queue_t queue;
    magma_int_t  dev = 0;
    magma_queue_create(dev, &queue);
```

```
6   magma_int_t m = ...; magma_int_t n = 1;

8   float *a; magma_smalloc_cpu(&a, m*m);
    float *b; magma_smalloc_cpu(&b, m);
10  float *x; magma_smalloc_cpu(&x, m);
    magma_int_t *piv = (magma_int_t *)malloc(m*sizeof(magma_int_t));
12
    //Belegen...
14
    const float alpha = 1.0f;
16  const float beta  = 0.0f;
    blasf77_sgemm("N", "N", &m, &n, &n, &alpha, a, &m, x, &m, &beta, b, &m);
18
    ////////////////////////////////////////////
20
    float *d_a; magma_smalloc(&d_a, m*m);
22  float *d_b; magma_smalloc(&d_b, m);
    magma_ssetmatrix(m, m, a, m, d_a, m, queue);
24  magma_ssetmatrix(m, n, b, m, d_b ,m, queue);

26  ////////////////////////////////////////////

28  magma_int_t info;
    magma_sgesv_gpu(m, n, d_a, m, piv, d_b, m, &info);
30
    ////////////////////////////////////////////
32
    magma_sgetmatrix(m, n, d_b, m, x, m, queue);
34  magma_free(d_a); magma_free(d_b);

36  ////////////////////////////////////////////

38  free(a); free(b); free(x); free(piv);

40  magma_queue_destroy(queue);
    magma_finalize();
42
```

Codebeispiel 6.29: MAGMA: GPU-Interface

Das GPU Interface funktioniert an sich ähnlich, es müssen lediglich Arrays explizit vom Host auf

das Device kopiert werden. Daher kann nun der CPU Speicher gewöhnlich mit magma_smalloc_cpu alloziert werden. GPU Speicher wird mit magma_smalloc erzeugt und mit magma_ssetmatrix bzw. magma_sgetmatrix kopiert. Der Aufruf der LU-Zerlegung magma_sgesv_gpu ist beinahe identisch. CPU Speicher wird normal mit free, GPU Speicher mit magma_free freigegeben.

MAGMA implementiert ebenfalls Algorithmen für dünne Matrizen. Aus bekannten Gründen wird hier jedoch nicht weiter darauf eingegangen.

Iterative Verfahren

Bei den Verfahren aus BLAS und LAPACK handelt es sich um direkte Lösungsverfahren. Diese liefern im Erfolgsfall (bis auf numerische Ungenauigkeiten) die analytische Lösung. Bei den sogenannten iterativen Verfahren handelt es sich um numerische Lösungen ähnlicher Probleme, deren Genauigkeit aber durch die Abbruchbedingung der Iteration gesteuert werden kann. Man opfert also Genauigkeit zu Gunsten von Performance.

Paralution

Paralution ist eine sehr umfangreiche C++-Bibliothek für indirekte (sparse) Lösungsverfahren von Differentialgleichungen oder Problemen der linearen Algebra. Im folgenden Beispiel soll die zweidimensionale stationäre Wärmeleitungsgleichung $\triangle x = b$ auf einem diskretisierten 100×100 Gitter gelöst werden. x bezeichnet die Temperatur, b eine Inhomogenität.

```
    paralution::init_paralution();
2   paralution::info_paralution();

4   paralution::LocalVector<float> x;
    paralution::LocalVector<float> rhs;
6
    paralution::LocalStencil<float> stencil(Laplace2D);
8   stencil.SetGrid(100);

10  x.Allocate("x", stencil.get_nrow()); x.Zeros();
    rhs.Allocate("rhs", stencil.get_nrow()); rhs.Ones();
12
    paralution::CG<LocalStencil<float>, LocalVector<float>, double> ls;
14  ls.SetOperator(stencil);
```

```
        ls.Build();
16      stencil.info();

18      ls.Solve(rhs, &x);

20      paralution::stop_paralution();
```

Codebeispiel 6.30: Paralution Beispiel

Die Daten werden hier mit Konstruktoraufrufen von C++-Vektoren erzeugt. Deren Inhalte werden durch explizite Aufrufe von Member Funktionen gesetzt. Dann wird ein sogenannter Stencil als numerische Näherung für den 2d-Laplace Operator erzeugt. Das Gitter, auf welches dieser Stencil angewendet werden soll, wird auf 100 (100×100) gesetzt. Damit wird ein linear Solver (ls) erzeugt (`ls.Build()`). Mit diesem kann die Differentialgleichung nun gelöst werden (`ls.Solve(rhs, &x)`). An Stelle eines Laplace Operators lässt sich genauso eine Matrix verwenden, um beispielsweise eine Zerlegung auszuführen. Mit der Initialisierunsfunktion `init_paralution` können mehrere Parameter, wie z.B. die Abbruchbedingung der Iteration eingestellt werden. Dies kann im äußerst ausführlichen Handbuch nachgeschlagen werden. [18]

AMGX

Bei AmgX handelt es sich um eine ähnliche Bibliothek. Die Stärken liegen hierbei u.A. bei Krylov-Subspace Methoden. AmgX ist ein offizielles Partnerprojekt von Nvidia und kann unter `https://github.com/NVIDIA/AMGX` heruntergeladen werden. Zusätzliche Python Bindings sind unter `https://github.com/shwina/pyamgx` verfügbar. Das Handbuch enthält mehr Informationen über die Nutzung. [3]

clSPARSE

Ein Teil von cuSPARSE und cuSOLVER sowie zusätliche iterative Verfahren wurden von AMD in OpenCL als clSPARSE implementiert. Das API wurde vollständig in C++ programmiert und ist kompatibel zum OpenCL C++ Wrapper. Die Bibliothek enthält folgende Funktionen:

- Sparse Matrix - dense Vector multiply (SpM-dV)
- Sparse Matrix - dense Matrix multiply (SpM-dM)
- Sparse Matrix - Sparse Matrix multiply Sparse Matrix Multiply(SpGEMM) - Single Precision
- Iterative conjugate gradient solver (CG)
- Iterative biconjugate gradient stabilized solver (BiCGStab)
- Dense to CSR conversions (& converse)
- COO to CSR conversions (& converse)
- Functions to read matrix market files in COO or CSR format

Folgendes Beispiel implementiert einen iterativen CG-Solver. Dazu werden eine dünne Matrix A sowie zwei dichte Vektoren x und b als C++-Objekte erstellt. Es folgt das Setup und der Handle, hier *Control* genannt.

```
    cldenseVector x;
2   clsparseInitVector(&x);

4   cldenseVector b;
    clsparseInitVector(&b);
6
    clsparseCsrMatrix A;
8   clsparseInitCsrMatrix(&A);

10  clsparseSetup();
    clsparseCreateResult createResult = clsparseCreateControl(queue());
12
```

Codebeispiel 6.31: clSPARSE: Initialisieren

Die Daten der C++ Klasse werden explizit gesetzt. Für die Grafikkarte muss dennoch Speicher alloziert und kopiert werden. Man beachte den eigenen Datentyp clsparseIdx_t. Das Format der Matrix (CSR) muss dem Handle bekannt gemacht werden („Meta“).

```
A.num_nonzeros = ...;
A.num_rows = ...;
A.num_cols = ...;
A.values =      cl::clCreateBuffer(context(), CL_MEM_READ_ONLY,
                                   A.num_nonzeros*sizeof(float));

A.col_indices = cl::clCreateBuffer(context(), CL_MEM_READ_ONLY,
                                   A.num_nonzeros*sizeof(clsparseIdx_t));

A.row_pointer = cl::clCreateBuffer(context(), CL_MEM_READ_ONLY,
                                  (A.num_rows + 1)*sizeof(clsparseIdx_t));

//Belegen und Kopieren...

clsparseCsrMetaCreate(&A, createResult.control);

x.num_values = A.num_cols;
x.values = clCreateBuffer(context(), CL_MEM_READ_ONLY,
                          x.num_values*sizeof(float));
clEnqueueFillBuffer(queue(), x.values, &number, sizeof(float), 0,
                    x.num_values*sizeof(float), 0, nullptr, nullptr);

b.num_values = A.num_rows;
b.values = clCreateBuffer(context(), CL_MEM_READ_WRITE,
                          b.num_values*sizeof(float));
clEnqueueFillBuffer(queue(), b.values, &number, sizeof(float), 0,
                          b.num_values*sizeof(float), 0, nullptr, nullptr);

```

Codebeispiel 6.32: clSPARSE: Vektoren und Matrizen setzen

Nach dem Setup kann ein eigener Handle für den Solver erstellt werden. Es wird ein diagonaler Vorkonditionierer verwendet. Die Konverkenzkriterien sind $\varepsilon_{rel} = 10^{-2}$ und $\varepsilon_{abs} = 10^{-5}$. Das Iterationslimit wird auf 1000 gesetzt. Schließlich wird die eigentliche Funktion gerufen. Beide Handles werden nach Gebrauch zerstört. Trotz C++ API müssen die Speicherobjekte freigegeben werden. Die Metainformation muss ebenfalls gelöscht werden.

```
    clsparseCreateSolverResult solverResult =
2     clsparseCreateSolverControl(DIAGONAL, 1000, 1e-2, 1e-5);

4   clsparseScsrcg(&x, &A, &b, solverResult.control, createResult.control);

6   clsparseReleaseSolverControl(solverResult.control);

8   clsparseReleaseControl(createResult.control);

10  clsparseTeardown();

12  clsparseCsrMetaDelete(&A);

14  clReleaseMemObject(A.values);
    clReleaseMemObject(A.col_indices);
16  clReleaseMemObject(A.row_pointer);

18  clReleaseMemObject(x.values);
    clReleaseMemObject(b.values);
20
```

Codebeispiel 6.33: clSPARSE: Ausführen

Die Dokumentation ist unter zu finden. Für das Error Handling muss neben dem üblichen *clSPARSE.h* zusätzlich *clSPARSE-error.h* inkludiert werden. Linken erfolgt gegen *libclSPARSE*.

Machine Learning

6.5. Machine Learning

Aus Wikipedia [19]:

Maschinelles Lernen ist ein Oberbegriff für die „künstliche“ Generierung von Wissen aus Erfahrung: Ein künstliches System lernt aus Beispielen und kann diese nach Beendigung der Lernphase verallgemeinern. Dazu bauen Algorithmen beim maschinellen Lernen ein statistisches Modell auf, das auf Trainingsdaten beruht. Das heißt, es werden nicht einfach die Beispiele auswendig gelernt, sondern es „erkennt“ Muster und Gesetzmäßigkeiten in den Lerndaten. So kann das System auch unbekannte Daten beurteilen [...].

Hauptanwendungsgebiete sind u.A. Klassifizierung, Filterung, Bild- und Spracherkennung, Regression oder Datenanalyse und Datenerzeugung. Man kann diese Algorithmen in drei Kategorien einteilen:

- **supervised Learning:** Eingabedaten durchlaufen ein Modell. Die Ausgabedaten werden mit einer Vorgabe verglichen. Aus der Differenz zwischen Ist- und Soll-Zustand kann ein Verlust ausgerechnet werden. Durch statistische Minimierung dieses Verlustes werden die Parameter des Modells angepasst. Falls es sich bei dieser Minimierung um eine auf Stochastik beruhender Methode handelt, spricht man von stochastischem Lernen.

- **reinforcement Learning:** Kann auch als supervised Learning angesehen werden. Statt einer Klassifikation der Ergebnisse in ”richtig” und ”falsch” wird dem Algorithmus hier eine Belohnung vorgegeben. Diese Methode synergiert am besten mit dem Lernverhalten des Menschen.

- **unsupervised Learning:** Der Algorithmus versucht selbständig zu den Eingangsdaten basierend auf den Charakteristika der Signale ein Modell zu erstellen. Die Anwendungsmöglichkeiten sind hier natürlich vergleichsweise begrenzt. Ein bekanntes Beispiel ist der EM-Algorithmus.

Zu den klassischen Algorithmen zählen z.B. Support Vector Machines (SVM), Clustering, Random Forest, Empirical Mode Decomposition (EMD), Principal Component Analysis (PCA), Independent Component Analysis (ICA), Single Spectrum Analysis (SSA), Informationstheorie und viele mehr. Diese beruhen im Wesentlichen auf einfacher Statistik in Kombination mit Methoden der linearen Algebra. SVMs beispielsweise berechnen Trennebenen zwischen linear separierbaren Punkten im n-dimensinalem Raum. Bei einem Random Forest handelt es sich im Grunde um eine Mittelung über verschiedene Entscheidungsbäume.

Allerdings erfreuen sich in modernen Anwendungen sogenannte neuronale Netze immer größerer Beliebtheit. Der folgende Abschnitt konzentriert sich lediglich auf supervised Learning mit neuronalen Netzen *off-line*, also Lernen, bei dem die Daten des trainierten Modells zu jedem Zeitpunkt erhalten bleiben.

neuronale Netze

Die Idee, ein menschliches Gehirn mit Computern nachzubauen, ist nicht neu und vermutlich so alt wie die Idee eines Computers. Die Anfänge lassen sich bereits auf die 40er Jahre zurückführen. Die ersten Theorien wurden in den 60ern entwickelt. Allerdings wurde die Entwicklung erst in den letzten Jahren von Erfolg gekrönt. Dies hat im Wesentlichen zwei Gründe. Der erste Grund ist *Big Data*: Durch den Aufstieg des Internets wurden internationale Konzerne wie Facebook oder Google zu Sammelbecken von Daten ungeahnten Ausmaßes. Wie bereits erwähnt ist eine gute Datengrundlage wichtig für das Training eines Modells. Diese Konzerne sind natürlich aus kommerziellen Gründen daran interessiert, Forschung in diesem Bereich zu finanzieren.

Der andere Grund beruht auf der Technik. Bei einem neuronalen Netz handelt es sich um ein SIMD Problem. Durch das Stagnieren der Performance von CPUs (siehe 2.1) und der Aufwändigkeit und Kosten von Rechenclustern, die für SIMD Probleme notwendig wären, sind CPUs für das Trainieren von Netzen kaum geeignet. Der Durchbruch von General-Purpose GPUs, sowohl auf hardware- als auch auch auf software-technischer Ebene, erlaubt es allerdings zu einem erschwinglichen Preis neuronale Netze auch bei kleinen Projekten mit kleinem Budget einzusetzen.

Neuronale Netze bestehen typischerweise aus einer Ausgabeschicht ("Output Layer") und mehreren verborgenen Schichten ("Hidden Layer"), die die Eingangssignale mit der Ausgabeschicht verbinden. Diese Schichten bestehen aus einer Vielzahl von Neuronen, die Eingangassignale mit verschiedenen Gewichten verrechnen. Eine logistische Funktion als Aktivierung entscheidet, ob der so berechnete Output an die nächste Schicht weitergegeben wird. Im Falle einer einzigen Schicht spricht man von einschichtigen Netzen (Single-Layer).

Man unterscheidet verschiedene Typen:

- **Single-Layer Perzeptron:** Ein einzelnes Neuron.
- **Multi-Layer Perzeptron:** Mehrere Perzeptronen hintereinander.
- **Fully-Connected Layer:** Jedes Neuron dieser Schicht gibt seinen Output an jedes Neuron der folgenden Schicht weiter.
- **Rekurrente Schicht:** Neuronen geben den Output auch an sich selbst weiter.
- **Convolutional Neural Network (CNN):** Vor der Klassifizierung durchlaufen die Daten eine Faltung, also Neuronen, die nur einen Teil des Inputs erhalten und auch nur einen Teil weitergeben.
- **Deep-Belief Network:** Eine Kombination von Boltzmann Maschinen.
- **Boltzmann Maschine:** Ein Stochastisches rekurrentes neuronales Netz zum Erlernen einer Wahrscheinlichkeitsverteilung.

  ...

Wichtigste moderne Anwendung ist wohl das CNN. Da Fully-Connected Netzwerke oft zu aufwändig sind, z.B. bei der Verarbeitung von 2d Daten wie Bildern, wird zunächst nur ein Teil des Inputs an ein Neuron weitergegeben. Dies kann z.B. ein quadratischer Ausschnitt aus einem Bild sein. Nach sogenanntem Pooling, dem Zusammenfassen mehrerer Neuronen (typischerweise 2×2) zu einer Aktivierung, wird der entstandene Output mit Fully-Connected Layern klassifiziert. Im mathematischen Sinne handelt es sich bei diesem Vorgang um eine Faltung. Dies motiviert den Begriff "convolutional", also "faltend".

Man betrachte ein Fully-Connected Network mit zwei Schichten aus drei und zwei Neuronen. Sei ein Input Signal $\vec{x} = (x_1, x_2, x_3)^T$ gegeben. Dann wird am ersten Neuron jeder dieser Inputs mit einem Gewicht multipliziert und die Summe $\Sigma = w_1^1 x_1 + w_2^1 x_2 + w_3^1 x_3$ an eine Aktivierungsfunktion weitergeleitet. Für das zweite und dritte Neuron findet dies genauso statt. Dieser Vorgang entspricht einer Matrixmultiplikation

$$\Sigma = (\Sigma_1, \Sigma_2, \Sigma_3)^T = W\vec{x} \tag{6.4}$$

wobei die Einträge von W die Gewichte $W_{ij} = w_j^i$ sind. Nach dieser Multiplikation mit einer 3×3 Matrix werden die Output Signale mit der Aktivierung verrechnet und an die nächste Schicht weitergegeben (feed-forward). Da es sich bei dieser nur um zwei Neuronen handelt, wird das dreikomponentige Signal mit einer 2×3 Multiplikation auf ein zweikomponentiges abgebildet. Dieses Signal wird nun mit dem Sollzustand verglichen. Aus der Differenz wird ein Verlust berechnet (mit einer Loss-Function). Auf Basis dessen werden die Gewichte mit einem Minimierungsverfahren rückwärts angepasst, um diesen Zustand besser abzubilden (Backpropagation). Dieser Vorgang wird iterativ wiederholt (Epochen), bis eine bestimmte Akkuratesse erreicht ist (Verhältnis richtiger Klassifikationen zur Gesamtzahl). Inwiefern diese Genauigkeit auf dem Trainingsset ein Indikator für das tatsächliche Verhalten ist, hängt von der Korrelation der Daten ab und muss daher mit einem Testset verifiziert werden. Test- und Trainingsset sollten ein repräsentativer Teil der Ausgangsdaten sein und keine Überschneidung aufweisen. Wenn das Modell sich zu gut auf die Trainingsdaten angepasst hat und eine deutlich schlechtere Genauigkeit auf den Testdaten liefert, so spricht man von Overfitting. CNNs gelten als besonders resistent gegen diesen Fehler.

Verbindet man im genannten Beispiel nur einen Input mit einem Neuron, so ergibt sich eine mathematische Faltung und die Gewichtsmatrix W reduziert sich auf $\mathrm{diag}(w_1^1, w_2^2, w_3^3)$. Bei dieser Matrix handelt es sich um eine sparse Matrix (und in diesem Spezialfall sogar um eine Band- und Diagonalmatrix).

cuDNN

[9]Bei cuDNN handelt es sich um die von Nvidia zur Verfügung gestellte GPU beschleunigte Implementierung von DNN Primitives. Dies sind die folgenden Routinen:

- Convolution (Forward und Backward), cross-Korrelation
- Pooling (Forward und Backward)
- Softmax (Forward und Backward)
- Neuron Aktivierungen (Forward und Backward):
  - Rectified linear (ReLU)
  - Sigmoid
  - Tangens Hyperbolicus (TANH)
- Tensor Transformationsfunktionen
- Local Response- und batch-Normalisierung, Locally-Connected Network (Forward und Backward)

Das folgende Beispiel zeigt eine einzelne Epoche eines CNN.

```
cv::Mat image = ...;

cudnnHandle_t cudnn;
cudnnCreate(&cudnn);

cudnnTensorDescriptor_t input_descriptor;
cudnnCreateTensorDescriptor(&input_descriptor);
cudnnSetTensor4dDescriptor(input_descriptor, CUDNN_TENSOR_NHWC,
  CUDNN_DATA_FLOAT, 1, 3, image.rows, image.cols);

cudnnTensorDescriptor_t output_descriptor;
cudnnCreateTensorDescriptor(&output_descriptor));
cudnnSetTensor4dDescriptor(output_descriptor,
  CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, 1, 3, image.rows, image.cols);
```

```
16    cudnnFilterDescriptor_t kernel_descriptor;
      cudnnCreateFilterDescriptor(&kernel_descriptor);
18    cudnnSetFilter4dDescriptor(kernel_descriptor,
        CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 3, 3, 3, 3);
20
      cudnnConvolutionDescriptor_t convolution_descriptor;
22    cudnnCreateConvolutionDescriptor(&convolution_descriptor);
      cudnnSetConvolution2dDescriptor(convolution_descriptor, 1, 1, 1, 1,
24      1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT);
```

Codebeispiel 6.34: cuDNN: Tensor-Descriptors

```
      cudnnConvolutionFwdAlgo_t convolution_algorithm;
2     cudnnGetConvolutionForwardAlgorithm(cudnn, input_descriptor,
        kernel_descriptor, convolution_descriptor,
4       output_descriptor, CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0,
        &convolution_algorithm);
6
      size_t workspace_bytes = 0;
8     cudnnGetConvolutionForwardWorkspaceSize(cudnn, input_descriptor,
        kernel_descriptor, convolution_descriptor,
10      output_descriptor, convolution_algorithm, &workspace_bytes);

12    void* d_workspace{nullptr};
      cudaMalloc(&d_workspace, workspace_bytes);
14
```

Codebeispiel 6.35: cuDNN: Konvolutions-Algorithmus

```
      int image_bytes = 1 * 3 * image.rows * image.cols * sizeof(float);
2     float* d_input{nullptr};
      cudaMalloc(&d_input, image_bytes);
4     cudaMemcpy(d_input, image.ptr<float>(0),
        image_bytes, cudaMemcpyHostToDevice);
6
      float* d_output{nullptr};
8     cudaMalloc(&d_output, image_bytes);
      cudaMemset(d_output, 0, image_bytes);
```

```
10
    const float kernel_template[3][3] = {
12    {1,  1, 1},
      {1, -8, 1},
14    {1,  1, 1}
    };
16
    float h_kernel[3][3][3][3];
18  for(int kernel = 0; kernel < 3; ++kernel)
    {
20    for (int channel = 0; channel < 3; ++channel)
      {
22      for (int row = 0; row < 3; ++row)
        {
24        for (int column = 0; column < 3; ++column)
          {
26        h_kernel[kernel][channel][row][column]
            = kernel_template[row][column];
28        }
        }
30    }
    }
32
    float* d_kernel{nullptr};
34  cudaMalloc(&d_kernel, sizeof(h_kernel));
    cudaMemcpy(d_kernel, h_kernel, sizeof(h_kernel), cudaMemcpyHostToDevice);
36
```

Codebeispiel 6.36: cuDNN: Kernel

```
    const float alpha = 1, beta = 0;
2   cudnnConvolutionForward(cudnn, &alpha, input_descriptor, d_input,
      kernel_descriptor,d_kernel, convolution_descriptor,convolution_algorithm,
4     d_workspace, workspace_bytes, &beta,output_descriptor, d_output);

6   float* h_output = new float[image_bytes];
    cudaMemcpy(h_output, d_output, image_bytes, cudaMemcpyDeviceToHost);
8
    ...
10
    delete[] h_output;
```

```
    cudaFree(d_kernel);
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_workspace);

    cudnnDestroyTensorDescriptor(input_descriptor);
    cudnnDestroyTensorDescriptor(output_descriptor);
    cudnnDestroyFilterDescriptor(kernel_descriptor);
    cudnnDestroyConvolutionDescriptor(convolution_descriptor);

    cudnnDestroy(cudnn);
```

Codebeispiel 6.37: cuDNN: feed forward

Exkurs: Python Tensorflow

[30]

https://www.tensorflow.org/overview/

[17]

```
    import tensorflow as tf
    mnist = tf.keras.datasets.mnist

    (x_train, y_train),(x_test, y_test) = mnist.load_data()
    x_train, x_test = x_train / 255.0, x_test / 255.0

    model = tf.keras.models.Sequential([
      tf.keras.layers.Flatten(input_shape=(28, 28)),
      tf.keras.layers.Dense(512, activation=tf.nn.relu),
      tf.keras.layers.Dropout(0.2),
      tf.keras.layers.Dense(10, activation=tf.nn.softmax)
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
      metrics=['accuracy'])

    model.fit(x_train, y_train, epochs=5)
```

```
    model.evaluate(x_test, y_test)
18
```

Codebeispiel 6.38: Keras/Tensorflow Beispiel

TensorRT

[31]

[32]

[1]

```
    from keras.models import load_model
2   import keras.backend as K
    from tensorflow.python.framework import graph_io
4   from tensorflow.python.tools import freeze_graph
    from tensorflow.core.protobuf import saver_pb2
6   from tenrflow.python.training import saver as saver_lib

8   def convert_keras_to_pb(keras_model, out_names, models_dir, model_filename):
      model = load_model(keras_model)
10    K.set_learning_phase(0)
      sess = K.get_session()
12    saver = saver_lib.Saver(write_version=saver_pb2.SaverDef.V2)

14    checkpoint_path = saver.save(sess, 'saved_ckpt',
        global_step=0, latest_filename='checkpoint_state')
16
      graph_io.write_graph(sess.graph, '.', 'tmp.pb')
18
      freeze_graph.freeze_graph('./tmp.pb', '',
20      False, checkpoint_path, out_names,
        "save/restore_all", "save/Const:0",
22      models_dir+model_filename, False, "")
```

Codebeispiel 6.39: Tensorflow Modell einfrieren

Cluster Computing mit NCCL

6.6. Cluster Computing mit NCCL

Kollektive Operationen

Die Notwendigkeit, Algorithmen für GPU-Cluster zu parallelisieren, motivierte die Implementierung der *Nvidia Collective Communications Library* (NCCL). Diese enthält mehrere Operationen, die auf Daten angewandt werden, die auf bis zu vier GPUs verteilt sind. Dazu wird zunächst ein spezieller Kommunikator definiert. Dann wird das entsprechende Gerät mit der jeweiligen Nummer ausgewählt. Für jedes Gerät wird ein Buffer der selben Größe erstellt sowie ein eigener Stream erstellt.

```
ncclComm_t comms[4];
int nDev = ...; int devs[nDev] = { 0, ... };

float** sendbuff = (float**)malloc(nDev*sizeof(float*));
float** recvbuff = (float**)malloc(nDev*sizeof(float*));
cudaStream_t* s = (cudaStream_t*)malloc(sizeof(cudaStream_t)*nDev);
for (int i = 0; i < nDev; ++i) {
  cudaSetDevice(i);
  cudaMalloc(sendbuff + i, size * sizeof(float));
  cudaMalloc(recvbuff + i, size * sizeof(float));
  cudaStreamCreate(s+i);
}
```

Codebeispiel 6.40: NCCL: Buffer und Streams

Nach Kopie der Daten kann der Kommunikator initialisiert werden. In diesem Fall werden die Nummern der Geräte in devs in das Kommunikator-Array eingetragen. Folgende kollektiven Operationen stehen zur Verfügung:

- ***All Reduce***: Eine Reduktion, bei der das Ergebnis in jedem Element des Outputarrays bereit steht.
- ***Broadcast***: Kopiert ein Array eines Geräts auf alle anderen.
- ***Reduce***: Wie *All Reduce*, aber das Ergebnis beschränkt sich auf den Outputbuffer eines einzigen Geräts.
- ***All Gather***: Auf jedem Gerät wird die selbe Anzahl an Werten von jedem Gerät aggregiert. Der Output wird nach Index des Geräts geordnet.
- ***Reduce Scatter***: Wie *Reduce*, aber das Ergebnis wird in Blöcken auf alle Geräte verteilt. Jedes Gerät erhält einen Teil der Daten basierend auf dessen Index.

```
    ncclCommInitAll(comms, nDev, devs);
2
    ncclGroupStart();
4   for(int i = 0; i < nDev; ++i)
    {
6     ncclAllReduce((const void*)sendbuff[i], (void*)recvbuff[i], size,
        ncclFloat, ncclSum, comms[i], s[i]);
8   }
    ncclGroupEnd();
10
```

Codebeispiel 6.41: NCCL: Multi-Device Reduktion

In diesem konkreten Beispiel wird für jedes Device die Maximumsreduktion auf den jeweiligen Buffern ausgeführt. Nach dem Aufruf von ncclGroupEnd steht das Gesamtergebnis in jedem Element der Outputbuffer bereit. Anschließend wird jedes Gerät synchronisiert.

```
    for(int i = 0; i < nDev; ++i)
2   {
      cudaSetDevice(i);
4     cudaStreamSynchronize(s[i]);
    }
6
    for(int i = 0; i < nDev; ++i)
8   {
      cudaSetDevice(i);
10    cudaFree(sendbuff[i]);
      cudaFree(recvbuff[i]);
12  }

14  for(int i = 0; i < nDev; ++i)
      ncclCommDestroy(comms[i]);
16
```

Codebeispiel 6.42: NCCL: Synchronisation und Cleanup

Neben der Freigabe der Buffer müssen zusätzlich die Kommunikatoren zerstört werden. Eine genauere Auflistung der Operationen sowie die Handhabung der Kommunikatoren kann im Handbuch https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html nachgelesen werden.

Device Abfrage mit MPI

Die genannten Operationen können nur auf maximal vier Geräte angewandt werden. Cluster bestehen jedoch aus hunderten GPUs. Typischerweise werden jeweils vier GPUs zusammengefasst und von einem Prozessor verwaltet. Zur Kommunikation dieser Prozessoren wir dann das sogenannte *Message Passing Interface* (MPI) angewandt. Dabei handelt es sich um eine ganz eigene Art der Parallelisierung und soll daher hier nicht tiefer behandelt werden. Im Prinzip führt man dasselbe Programm auf allen Prozessoren unter Angabe von Hostnamen oder Nodes aus. In diesem Programm wird nach Initialisierung jedem Prozess eine Nummer („Rank“) zugeordnet. Der nachfolgende Code wir dann von jedem Prozessor exakt zeitgleich ausgeführt. So kann jeder Prozessor seine Devices nacheinander unabhängig von den anderen abfragen.

```
MPI_Init(...);

int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

int deviceCount;
cudaGetDeviceCount(&deviceCount);
cudaDeviceProp deviceProp;
for(uint device = 0; device < deviceCount; ++device)
{
  cudaGetDeviceProperties(&deviceProp, device);
  printf("Host %s with rank %d (of %d) has %d device(s).
    Its device #%d is %s and has compute capability %d.%d.\n",
    processor_name, world_rank, world_size, deviceCount,
    device+1, deviceProp.name, deviceProp.major, deviceProp.minor);
}

MPI_finalize();

```

Codebeispiel 6.43: Device Abfrage mit MPI

Über den Rank kann ein Prozessor eine einzelne Aufgabe erfüllen, die nur für ihn bestimmt ist. Beispielsweise können zwei Prozesse direkt Daten miteinander austauschen. Mit einem GPU-Build von MPI ist es sogar möglich, explizit GPU-Memory zu senden. Dazu erstellt ein Prozess (0) einen Buffer für die Daten auf der Grafikkarte. Der andere Prozess (1) erstellt einen Buffer der selben Größe im Speicher seiner Grafikkarte. Prozess 0 kopiert die Daten. Dann sendet er explizit diese Daten an Rank 1. Rank 1 muss diese explizit abnehmen und speichert diese dabei in seinem Buffer ab. Zum Schluss können die Daten auf den neuen Host kopiert werden.

```
MPI_Init(...);

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int size = ...;
if(rank == 0) { int *send_buf; cudaMalloc(&send_buf, size*sizeof(int)); }
if(rank == 1) { int *recv_buf; cudaMalloc(&recv_buf, size*sizeof(int)); }

...

if(rank == 0)
  cudaMemcpy(send_buf, ..., size*sizeof(int), cudaMemcpyHostToDevice);
MPI_Send(send_buf, size, MPI_INT, 1, 0, MPI_COMM_WORLD);
...
MPI_Recv(recv_buf, size, MPI_INT, 0, 0, MPI_COMM_WORLD,
  MPI_STATUS_IGNORE);
if(rank == 1)
  cudaMemcpy(..., recv_buf, size*sizeof(int), cudaMemcpyDeviceToHost);

...

MPI_finalize();
```

Codebeispiel 6.44: Austausch von Device Memory mit MPI

GPU-Cluster

Auf einem GPU-Cluster werden typischerweise beide Methoden kombiniert. Ein einzelnes Programm wird auf beliebig vielen Hosts ausgeführt. Die einzelnen Prozesse können mittels MPI explizit kommunizieren. Jeder dieser Prozesse kann frei zwischen bis zu vier Grafikkarten wählen und parallele Algorithmen unter Zuhilfenahme von NCCL ausführen. MPI ermöglicht das freie Senden der Daten ohne zusätzliches Buffering im Hostmemory. Die GPUs können mit PCIe oder NVlink angebunden sein. Im letzteren Fall sogar direkt untereinander. Die Prozessoren sind in einem Highspeed-Netzwerk (z.B. mit Infiniband) miteinander verbunden. Da diese Prozessoren normalerweise über mehrere Kerne verfügen, können diese, während die GPUs

beschäftigt sind, mittels thread-parallelem Programmieren zusätzlich Aufgaben erledigen. So kann die Hardware vollständig ausgelastet werden.

Das NCCL-Handbuch gibt unter
https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/mpi.html
Empfehlungen ab zur Verwendung von MPI mit NCCL. Ein Beispiel dafür ist unter
https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/examples.html#example-3-multiple-devices-per-thread
zu finden.

6.7. Graph-Computing mit nvGRAPH

https://docs.nvidia.com/cuda/nvgraph/index.html

OpenACC v.s. OpenMP

7. OpenACC v.s. OpenMP

Um die bisher genannten Konzepte zu vereinfachen, existieren seit einigen Jahren Bestrebungen, Compiler zu entwickeln, die ohne das explizite Programmieren von Kernel bestimmte Teile des Programms automatisch in parallelen Maschinencode übersetzen. OpenMP ist eine Library für thread-paralleles Programmieren angewandt auf Systeme mit shared Memory, also im Wesentlichen Prozessoren mit mehreren Kernen und gemeinsamen Speicher. 2012 haben sich einige Programmierer für ein neues Projekt abgespaltet: OpenACC ist ein Programmierstandard der *Portland Group*, der ein ähnliches Konzept verfolgt. Mittels Compileranweisungen (#pragma) zeichnet man Regionen zur Parallelisierung aus und gibt lediglich Hinweise für Optimierungen an. Der Compiler versucht dann eigenständig diesen Teil des Codes plattformunabhängig wahlweise in paralleles Assembly oder NVPTX zu übersetzen. Das Projekt läuft unter dem Slogan *More Science, Less Programming*. Der Programmierer soll mit HPC möglichst wenig konfrontiert werden und damit mehr Zeit für die Entwicklung wissenschaftlicher Applikationen haben. Dieser Gewinn an Funktionalität ist allerdings nur auf Kosten von Performance möglich. Selbst die Entwickler gehen von einem Leistungsverlust gegenüber CUDA von etwa 30% aus. Die Portland Group wurde 2013 von der *Nvidia Corporation* gekauft.

Seit Version 4.5 unterstützt OpenMP ebenfalls Offloading, also das Verschieben von Daten in den Speicher der GPU.

7.1. OpenACC Direktiven mittels PGI-SDK

Nur wenige Compiler unterstützen OpenACC. Lediglich der C++-Compiler der Portland Group unterstützt den Standard vollständig. Es handelt sich dabei um einen kommerziellen Compiler. Eine kostenlose Version des PGI-SDK ist aber unter `https://www.pgroup.com/products/community.htm` verfügbar.

Loops

Zunächst wird ein Teil des Codes, der als parallelisierbar erkannt wurde, mit einem Pragma und einem extra Scope ausgezeichnet. Es folgt ein spezielles Pragma für die Parallelisierung einer Schleife innerhalb dieser Scope.

```
    #pragma acc parallel
2   {
      #pragma acc loop
4     for(int i = 0; j < N; ++i) a[i] = 0;
    }
6
```

Codebeispiel 7.1: OpenACC: Loops

Die gekennzeichnete Schleife wir dann automatisch auf alle verfügbaren Threads aufgeteilt. Mittels `#pragma acc parallel loop reduction(max:err)` lassen sich Variablen für eine Reduktion angeben. Nach Abhandlung der Schleife steht das Ergebnis in `err` bereit. Tabelle 7.1 zeigt eine Auflistung aller implementierten Reduktionsoperationen.

Operator	Description	Example
+	Addition/Summation	reduction(+:sum)
*	Multiplication/Product	reduction(*:product)
max	Maximum value	reduction(max:maximum)
min	Minimum	valuereduction(min:minimum)
&	Bitwise and	reduction(& :val)
|	Bitwise or	reduction(|:val)
&&	Logical and	reduction(&& :val)
||	Logical or	reduction(||:val)

Tabelle 7.1.: Reduktionen in OpenACC

Speicherallozierungen

Zur Berechnung mittels GPU müssen Daten in den Speicher der GPU verschoben werden. Um dies zu umgehen, kann pinned Memory verwendet werden. Dies muss beim Kompilieren jedoch

angegeben werden (siehe 7.1). Vor Eintreten in eine parallele Region gibt man dem Compiler einen Hinweis, welche Arrays auf die GPU kopiert werden sollen (copyin) und welche erstellt werden müssen (create). Am Ende der Region muss angegeben werden, welche Daten die GPU verlassen (copyout) und welcher Speicher freigegeben wird (delete). Dabei unterstützt OpenACC Arrayslicing ähnlich Python:
copyin(array[starting_index:length])
copy erstellt Speicher (falls nötig) und kopiert dann explizit.

```
#pragma acc enter data copyin(a[0:N],b[0:N]) create(c[0:N])

#pragma acc parallel loop
for(int i = 0; i < N; ++i) c[i] = a[i] + b[i];

#pragma acc exit data copyout(c[0:N]) delete(a,b)
```

Codebeispiel 7.2: OpenACC: Speicherverwaltung

Ein bekanntes Beispiel ist die Lösung der stationären 2d Wärmeleitungsgleichung. Jeder neue Gitterpunkt ist der Mittelwert der Nachbarpunkte:

```
...

float error;

#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])
while(err > tol && iter < iter_max)
{
  error = 0.0f;

  #pragma acc parallel loop reduction(max:err)
    copyin(A[0:n*m]) copy(Anew[0:n*m])
  for(intj = 1; j < n-1; ++j)
  {
    for(int i = 1; i < m-1; ++i)
    {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +A[j-1][i] + A[j+1][i]);
```

```
18            error = fmax(err, abs(Anew[j][i] - A[j][i]));
            }
20        }


22

        #pragma acc parallel loop copyin(Anew[0:n*m]) copyout(A[0:n*m])
24      for(int j = 1; j < n-1; ++j)
        {
26        for(int i = 1; i < m-1; ++i) A[j][i] = Anew[j][i];
        }
28

        iter++;
30    }

32    ...
```

Codebeispiel 7.3: OpenACC: Wärmeleitungsgleichung

Die while-Schleife muss ebenfalls markiert werden, anderfalls wird nach jeder Iteration überflüssigerweise kopiert.

Zur Synchronisation existiert die update Direktive, die eine Barriere bildet, bis bestimmte Daten bereit stehen (device für GPU, self für CPU):

```
update self(x[0:count])

update device(x[0:count])
```

Kompilierung und Profiling

Es existieren zwei verschiedene Compiler für C und C++:

```
pgcc  -fast -Minfo=all -ta=tesla main.c

pgc++ -fast -Minfo=all -ta=tesla main.cpp
```

-fast steht für die schnellste Optimierung, -Minfo=all für die meisten Warnungen und Informationen zur Parallelisierung. Die wichtigste Angabe ist aber die des Target Accelerators. -ta=tesla steht dabei für eine Tesla GPU (-ta=tesla:managed für pinned Memory), -ta=multicore für einen beliebigen Mehrkernprozessor. Die Ausführbare Datei lässt sich mit dem PGI Profiler genauer untersuchen:

```
pgprof ./a.out
```

Dieser untersucht nicht nur die OpenACC Anweisungen, sondern gibt auch die zugrunde liegenden CUDA Funktionen und deren Laufzeit aus. Dieses Tool entspricht nvprof.

Structs und Klassen

Befehle zum Kopieren oder zum Löschen lassen sich auch direkt in einer Datenstruktur verbauen.

```
typedef struct
{
  float* arr;
  int n;
} vector;

int main(void)
{
  vector v;
  v.n = 10;
  v.arr = (float*) malloc(v.n*sizeof(float));
  #pragma acc enter data copyin(v)
  #pragma acc enter data create(v.arr[0:v.n])
  ...
  #pragma acc exit data delete(v.arr)
  #pragma acc exit data delete(v)
  free(v.arr);
}
```

Codebeispiel 7.4: OpenACC: C-Structs

In C++ können diese Anweisungen sogar im Konstruktor oder Destruktor einer Klasse verwendet werden.

```
class vector
{
private:
  float* arr;
  int n;
public:
  vector(int size)
  {
    n = size;
    arr = new float[n];
    #pragma acc enter data copyin(this)
    #pragma acc enter data create(arr[0:n])
  }

  ~vector()
  {
    #pragma acc exit data delete(arr)
    #pragma acc exit data delete(this)
    delete(arr);
  }
};

```

Codebeispiel 7.5: OpenACC: C++-Klassen

Aus design-technischen Gründen (Encapsulation) empfiehlt es sich, für ein update-Pragma einen Wrapper als Member Funktion zu programmieren:

```
void accUpdateSelf()   { #pragma acc update   self(arr[0:n]) }
void accUpdateDevice() { #pragma acc update device(arr[0:n]) }
```

Codebeispiel 7.6: OpenACC: Update Member-Funktion

Fortgeschrittenes

Folgen zwei Schleifen aufeinander, was oft bei mehrdimensionalen Daten der Fall ist, lässt sich eine bestimmte Anzahl davon zu einer zusammenfügen:

```
#pragma acc parallel loop collapse(2)
for(int i = 0; i < 4; ++i)
  for(j = 0; j < 4; ++j)
    array[i][j] = 0.0f;
```

Codebeispiel 7.7: OpenACC: Loop Collapse

Tiles entsprechen in etwa einem 2D Block in CUDA:

```
#pragma acc kernels loop tile(2,2)
for(int x = 0; x < 4; ++x)
{
  for(int y = 0; y < 4; ++y)
  {
    array[x][y]++;
  }
}
```

Codebeispiel 7.8: OpenACC: Tile

Blöcke in CUDA entsprechen sogenannten „Gangs“ in OpenACC. Warps entsprechen „Workers“, Threads entsprechen „Vectors“. Für alle drei lässt sich explizit eine Zahl angeben. Dann kann eine Schleife für eine dieser Gruppen markiert werden.

```
#pragma acc parallel num_gangs(2) num_workers(2) vector_length(32)
{
  #pragma acc loop gang worker
  for(int x = 0; x < 4; ++x)
  {
    #pragma acc loop vector
    for(int y = 0; y < 32; ++y) array[x][y]++;
  }
}
```

Codebeispiel 7.9: OpenACC: Gangs Workers Vectors

7.2. OpenACC und CUDA

Bei OpenACC geht es um die einfache Parallelisierung von eigenem Code ohne die explizite Programmierung von Kernel. In den meisten Fällen ist man jedoch auf zusätzliche CUDA Librarys angewiesen. Um eine CUDA Funktion mit einem durch ein Pragma kopiertes Array im Device Speicher aufzurufen, muss dieses extra angegeben werden, da der Variablenname lediglich ein Pointer auf Host Memory ist.

```
float* x = ...;
int n = ...;
#pragma acc data copyin(x[0:n])
{
  ...
  #pragma acc host_data use_device(x)
  <Cuda-Funktion>(x);
  ...
}
```

Codebeispiel 7.10: OpenACC/CUDA Zusammenspiel: Host Memory

Im umgekehrten Fall muss ein Device Pointer für eine Schleife angegeben werden. So kann

beispielsweise ein von einer CUDA Funktion modifiziertes Array in OpenACC wie gewohnt ohne eine Kopie benutzt werden.

```
float* x;
int n = ...;
cudaMalloc(&x, n*sizeof(float));
...
#pragma acc kernels deviceptr(x)
for(int i = 0; i < n; ++i) x[i] = i;
```

Codebeispiel 7.11: OpenACC/CUDA Zusammenspiel: Device Memory

7.3. OpenMP und Offloading

Das Beispiel Wärmeleitungsgleichung könnte in OpenMP etwa so aussehen:

```
...
float error;
#pragma omp target data map(alloc:Anew) map(A)
while(error > tol && iter < iter_max)
{
  error = 0.0f;
  #pragma omp target teams distribute parallel for reduction(max:error)
  for(int j = 1; j < n-1; ++j)
  {
    for(int i = 1; i < m-1; ++i)
    {
      Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] + A[j-1][i] + A[j+1][i]);
      error = fmax(error, fabs(Anew[j][i] - A[j][i]));
    }
  }


```

```
20      #pragma omp target teams distribute parallel for
        for(int j = 1; j < n-1; ++j)
22      {
          for(int i = 1; i < m-1; ++i) A[j][i] = Anew[j][i];
24      }

26      iter++;
    }
28  ...
```

Codebeispiel 7.12: OpenMP: Offloading

OpenCL auf FPGAs

8. High Performance Computing mit FPGAs

8.1. Field Programmable Gate Array

Aus Wikipedia: "Ein Field Programmable Gate Array (FPGA) ist ein integrierter Schaltkreis (IC) der Digitaltechnik, in welchen eine logische Schaltung geladen werden kann. Die englische Bezeichnung kann übersetzt werden als im Feld (also vor Ort, beim Kunden) programmierbare (Logik-)Gatter-Anordnung.

Anders als bei der Programmierung von Computern, Microcontrollern oder Steuerungen bezieht sich hier der Begriff Programmierung nicht nur auf die Vorgabe zeitlicher Abläufe, sondern vor allem auf die Definition der gewünschten Schaltungsstruktur. Diese wird mittels einer Hardwarebeschreibungssprache formuliert und von einer Erzeugersoftware in eine Konfigurationsdatei übersetzt, welche vorgibt, wie die physischen Elemente im FPGA verschaltet werden sollen. Man spricht daher auch von der Konfiguration eines FPGA."

Seit einigen Jahren werden FPGAs nicht nur im Bereich Elektronik, sondern auch vermehrt für Anwendungen im HPC verwendet. Dabei werden Steckkarten für die PCIe Schnittstelle entwickelt, die mit verschiedenen Programmiersprachen vom Host aus programmiert werden und bestimmte Teile eines Programms auf einem FPGA als Koprozessor beschleunigen. Dieses Programm wird von einem speziellen Compiler, der vom Hersteller entwickelt wurde, in sogenannte Hardware Beschreibungssprache (HDL) übersetzt. FPGAs werden meistens bei schwer parallelisierbaren Problemen eingesetzt, z.B. in der Finanzmathematik, bei Datenbankzugriffen und bei Dateikompression. Allerdings versuchen Hersteller dies auch auf SIMD Probleme zu erweitern. Vorreiter ist hier der Marktführer Xilinx, der auch Librarys für Machine Learning anbietet, wie etwa eine FPGA Implementierung von Tensorflow. Xilinx verspricht einen bis zu

achtfach höheren Durchsatz beim Evaluieren eines trainierten Modells gegenüber einer Tesla V100.

Im Folgenden soll am Beispiel Vektoraddition das Programmiermodell verdeutlicht werden. Das Programm wurde für eine Alveo Karte von Xilinx geschrieben.

8.2. Offloading mittels OpenCL API

Die quelloffene OpenCL C/C++ API lässt sich gut für das Offloading von Speicher und das Ausführen von Kernel auf dem FPGA nutzen. Der Hersteller stellt dafür einen eigenen Treiber bereit. Die Programmierung des Host Codes funktioniert damit analog zu OpenCL. Ein Array im Host Speicher muss allerdings page-aligned abgelegt werden, um bei der Kopie in den global Memory überflüssiges Umkopieren zu vermeiden. Alle benötigten Funktionen befinden sich im Header *xcl2.hpp* unter https://github.com/Xilinx/SDAccel_Examples/tree/master/libs/xcl2, der weitere Xilinx spezifische Dateien inkludiert. Zur Benutzung muss beim Kompilieren mit *libxilinxopencl* gelinkt werden, eine Erweiterung für OpenCL. Header und Librarys sind Teil der quelloffenen Xilinx Runtime (XRT), die unter https://github.com/Xilinx/XRT heruntergeladen werden kann.

```
#include "xcl2.hpp"
#define DATA_SIZE 4096
...

std::vector<int, aligned_allocator<int>> source_in1(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_in2(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_hw_results(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_sw_results(DATA_SIZE);

...

auto devices = xcl::get_xil_devices();
auto device = devices[0];

cl::Context context(device, NULL, NULL, NULL, NULL);
cl::CommandQueue q(context, device, CL_QUEUE_PROFILING_ENABLE, NULL);

std::string binaryFile = ...:
unsigned fileBufSize;
```

```
20    auto fileBuf = xcl::read_binary_file(binaryFile, fileBufSize);
      cl::Program::Binaries bins{{fileBuf, fileBufSize}};
22
      devices.resize(1);
24    cl::Program program(context, devices, bins, NULL, NULL);
      cl::Kernel krnl_vector_add(program, "vadd", NULL);
26
      cl::Buffer buffer_in1(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
28      DATA_SIZE*sizeof(int), source_in1.data(), NULL);
      cl::Buffer buffer_in2(context, CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
30      DATA_SIZE*sizeof(int), source_in2.data(), NULL);
      cl::Buffer buffer_output(context,CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
32      DATA_SIZE*sizeof(int), source_hw_results.data(), NULL);

34    krnl_vector_add.setArg(0, buffer_in1);
      krnl_vector_add.setArg(1, buffer_in2);
36    krnl_vector_add.setArg(2, buffer_output);
      krnl_vector_add.setArg(3, DATA_SIZE);
38
      q.enqueueMigrateMemObjects({buffer_in1, buffer_in2}, 0);
40
      q.enqueueTask(krnl_vector_add);
42
      q.enqueueMigrateMemObjects({buffer_output}, CL_MIGRATE_MEM_OBJECT_HOST);
44    q.finish();
      delete[] fileBuf;
46
      ...
48
```

Codebeispiel 8.1: FPGA: Host Programm

Da es sich bei *xcl2.hpp* um einen C++ Header handelt, sollte auch die OpenCL C++ Wrapper API verwendet werden. Besonders sind an diesem Programm lediglich zwei Dinge. Die Funktion xcl::get_xil_devices sucht mit den üblichen Abfragen nach einem Gerät, das "Xilinx" im Herstellernamen trägt. Die Funktion xcl::read_binary_file liest eine ausführbare Datei ein. Bei dieser handelt es sich um ein cross-platform kompiliertes Binary, dass für eine ganz bestimmte FPGA Karte erzeugt wurde.

Die Funktion enqueueTask führt ein Kernel mit der Workgroup Größe (1,1,1) aus.

8.3. Kernels in OpenCL und C/C++

Kernel lassen sich in C, C++, OpenCL oder HDL schreiben. Bei einer HDL handelt es sich nicht um eine Programmiersprache und stellt damit ein völlig anderes Konzept dar. Da dieses Thema sehr komplex ist, soll hier nicht darauf eingegangen werden. Die einfachste Möglichkeit stellt ein Programm in C/C++ dar. Es handelt sich hierbei um sogenannte High-Level Synthesis (HLS), also das Erzeugen von HDL aus einer höheren Programmiersprache.

```
#define BUFFER_SIZE 1024
#define DATA_SIZE 4096
const unsigned int c_len = DATA_SIZE / BUFFER_SIZE;
const unsigned int c_size = BUFFER_SIZE;
extern "C" {
void vadd(const int *in1, const int *in2, int *out_r, const int size)
{
  #pragma HLS INTERFACE m_axi port = in1 offset = slave bundle = gmem
  #pragma HLS INTERFACE m_axi port = in2 offset = slave bundle = gmem
  #pragma HLS INTERFACE m_axi port = out_r offset = slave bundle = gmem
  #pragma HLS INTERFACE s_axilite port = in1 bundle = control
  #pragma HLS INTERFACE s_axilite port = in2 bundle = control
  #pragma HLS INTERFACE s_axilite port = out_r bundle = control
  #pragma HLS INTERFACE s_axilite port = size bundle = control
  #pragma HLS INTERFACE s_axilite port = return bundle = control

  int v1_buffer[BUFFER_SIZE];
  int v2_buffer[BUFFER_SIZE];
  int vout_buffer[BUFFER_SIZE];

  for (int i = 0; i < size; i += BUFFER_SIZE)
  {
  #pragma HLS LOOP_TRIPCOUNT min=c_len max=c_len
    int chunk_size = BUFFER_SIZE;
    if((i + BUFFER_SIZE) > size) chunk_size = size - i;

    read1: for(int j = 0; j < chunk_size; j++)
    {
    #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
    #pragma HLS PIPELINE II=1
      v1_buffer[j] = in1[i + j];
    }

```

```
36          read2: for(int j = 0; j < chunk_size; j++)
            {
38          #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            #pragma HLS PIPELINE II=1
40            v2_buffer[j] = in2[i + j];
            }
42
            vadd: for(int j = 0; j < chunk_size; j++)
44          {
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
46          #pragma HLS PIPELINE II=1
              vout_buffer[j] = v1_buffer[j] + v2_buffer[j];
48          }

50          write: for(int j = 0; j < chunk_size; j++)
            {
52          #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            #pragma HLS PIPELINE II=1
54            out_r[i + j] = vout_buffer[j];
            }
56        }
        }}
58
```

Codebeispiel 8.2: FPGA: C Kernel

Der offensichtliche Unterschied besteht in den Pragma Anweisungen. Da dieses Programm rein seriell abläuft, muss man mittels Pragmas den Compiler anweisen, die Schleifen auszurollen und damit parallele Hardware zu erzeugen. Die verschiedenen Pragmas können im SdAccel Pragma Guide [29] nachgelesen werden. Der Compiler erzeugt hier also eine Logik für jedes Vektorelement. Diese Schleifen lassen sich benennen, um im Synthesereport aufgelistet zu werden. Dazu muss man jedoch das kostenlose auf Eclipse basierende SdAccel Tool von Xilinx nutzen (Nachfolger 2020: Scout). Die Handhabung dessen wird im Handbuch erklärt. [27]

Um unnötige Zugriffe zu vermeiden, teilt man die Daten in Portionen von vier Kilobyte (page size) ein. Lese-, Rechen- und Schreiboperationen werden getrennt audsgeführt und in Buffer geschrieben. Dieses Verfahren nennt sich Burst Access.

Dieses Kernel lässt sich fast analog in OpenCL implementieren.

```
#define BUFFER_SIZE 1024
#define DATA_SIZE 4096

__constant uint c_len = DATA_SIZE/BUFFER_SIZE;
__constant uint c_size = BUFFER_SIZE;

kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void vadd(__global const int *in1, __global const int *in2,
  __global int *out_r, const int size)
{
  int v1_buffer[BUFFER_SIZE];
  int v2_buffer[BUFFER_SIZE];

  __attribute__((xcl_loop_tripcount(c_len, c_len)))
  for (int i = 0 ; i < size; i += BUFFER_SIZE)
  {
    int chunk_size = BUFFER_SIZE;

    if(i + chunk_size > size) chunk_size = size - i;

    __attribute__((xcl_loop_tripcount(c_size, c_size)))
    __attribute__((xcl_pipeline_loop(1)))
    read1: for(int j = 0 ; j < chunk_size; ++j) v1_buffer[j] = a[i+j];

    __attribute__((xcl_loop_tripcount(c_size, c_size)))
    __attribute__((xcl_pipeline_loop(1)))
    read2: for (int j = 0 ; j < chunk_size; ++j) v2_buffer[j] = b[i+j];

    __attribute__((xcl_loop_tripcount(c_size, c_size)))
    __attribute__((xcl_pipeline_loop(1)))
    vadd: for (int j = 0 ; j < chunk_size; ++j)
      out_r[i+j] = v1_buffer[j] + v2_buffer[j];
  }
}
```

Codebeispiel 8.3: FPGA: OpenCL Kernel

8.4. Xilinx Compiler

Während das Host Programm wie gewohnt mit einem beliebigen C oder C++ Compiler übersetzt wird, muss das FPGA Binary mit einem Compiler von Xilinx erstellt werden. Dieser heißt v++ und ist Teil der *Vitis* Platform. Der Compiler übersetzt zunächst den Device Code unter Angabe der Kernelfunktion und der Karte in eine Objektdatei mit Dateiendung .xo. Danach erfolgt das Linken und Erstellen einer ausführbaren Datei mit Dateiendung .xclbin.

```
v++ -c --target ... --kernel vadd --platform xilinx_u250_qdma_201910_1 vadd.cpp -o vadd.xo

v++ -l --platform xilinx_u250_qdma_201910_1 vadd.xo -o vadd.xclbin --target ...
```

Es gibt drei mögliche Angaben für --target.

sw_emu bezeichnet die Softwareemulation. Dabei wird mit einem gewöhnlichen Compiler gewöhnlicher Host Code erzeugt um die funktionale Richtigkeit des Programms zu testen.

hw_emu bezeichnet die Hardwareemulation. Dabei wird mit einem gewöhnlichen Compiler die Hardwareerzeugung lediglich in Software auf dem Host simuliert. Die Hardwareemulation ist ein erster Indikator für die Funktionsfähigkeit des Programms in Hardware.

hw bezeichnet das tatsächliche Programm für das FPGA. Es sollte erst nach der Hardwareemulation erzeugt werden, da das Kompilieren extrem lange dauert. Die Vektoraddition beispielsweise benötigte mit einem Prozessor mit 24 Kernen fast zwei Stunden zum Erzeugen. Sollte die Hardwareemulation bereits fehlgeschlagen sein, ist der Versuch ein echtes Programm zu erzeugen Zeitvergeudung.

Weitere Compileroptionen lassen sich im Handbuch der *Xilinx Commandline Tools* nachlesen. [28] Dieses ist noch für den alten xocc-Compiler verfasst, sollte aber analog für v++ gelten.

8.5. Xilinx Alveo und Librarys

Bei den verwendeten Boards Alveo U200, U250, U280 und U50 handelt es sich derzeit um die fortschrittlichsten ihrer Art. Mehr Informationen lassen sich im Datenblatt finden. [23]

Aufgrund von sehr speziellen Eigenschaften, beispielsweise ungewöhnliche Integer Arithmetik, stellt Xilinx für verschiedene Bereiche eigene Bibliotheken bzw. Reimplementierungen bekannter Librarys zur Verfügung. Eine genauere Beschreibung befindet sich in Kapitel 2 im Handbuch der Xilinx HLS. [38]

Weitere HPC Librarys wurden für diese Art von Boards ertellt. Siehe unter Anderem:

- Xilinx OpenCV
  https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1233-xilinx-opencv-user-guide.pdf
- Machine Learning Suite
  https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html
- BlackLynx High Speed Search
  https://www.xilinx.com/products/acceleration-solutions/1-zzrgcy.html
- Falcon Accelerated Genomics Pipelines
  https://www.xilinx.com/products/acceleration-solutions/1-zzroc0.html
- Maxeler Real Time Risk
  https://www.xilinx.com/products/acceleration-solutions/1-ykfnpg.html

9. Intel oneAPI

Neuere Entwicklungen im HPC Bereich gehen in Richtung von hardwarenahem heterogenem Rechnen. Für 2020 ist der Release von neuen Grafikkarten der Firma Intel unter dem Codenamen „Xe“ geplant. Die Firma ist eigentlich bekannt für die Herstellung und Vermarktung von High-End Prozessoren und Serveranlagen. Dafür existiert eine Reihen von kommerziellen Produkten, die speziell für Intel-Chips ausgelegt sind, z.B. ein eigener Compiler, eine MPI-Implementierung sowie die *Intel Math Kernel Library* (MKL). Zudem wurde 2015 der zweitgrößte FPGA-Produzent Altera gekauft. Um diese Sparten nun zu verbinden, fördert Intel zur Zeit die Entwicklung von *oneAPI*. Dabei handelt es sich um ein High-Level C++-API, das auf *Data Parallel C++* (dpcpp) und *Sycl* aufbaut. Letzteres ist selbst ein C++API für OpenCL.

Ein Programm besteht aus drei verschiedenen Teilen. Die sogenannte *Application Scope* ähnelt stark OpenCL. Dabei werden auf dem Host Plattform und Device ausgewählt sowie Buffer erstellt. Dann wird eine Command Queue erstellt. Mit dem Befehl submit werden dieser ein Handle, eine Folge von Bufferkopien und entsprechende Modifikatoren sowie der Kernelcode übergeben.

```
#include<vector>
#include<CL/sycl.hpp>
#define SIZE ...

namespace sycl = cl::sycl;

...

  std::array<int, SIZE> a, b, c;
  ...

  sycl::range<1> a_size{SIZE};
```

```
14    auto platforms = sycl::platform::get_platforms();
      for(auto &platform : platforms) auto devices = platform.get_devices();
16
      sycl::default_selector device_selector;
18    sycl::queue d_queue(device_selector);

20    sycl::buffer<int, 1>  a_device(a.data(), a_size);
      sycl::buffer<int, 1>  b_device(b.data(), a_size);
22    sycl::buffer<int, 1>  c_device(c.data(), a_size);

24    d_queue.submit(
        [&](sycl::handler &cgh)
26      {
          auto c_res = c_device.get_access<sycl::access::mode::write>(cgh);
28
          auto a_in = a_device.get_access<sycl::access::mode::read>(cgh);
30        auto b_in = b_device.get_access<sycl::access::mode::read>(cgh);

32        cgh.parallel_for<class ex1>(
            a_size, [=](sycl::id<1> idx)
34          {
              c_res[idx] = a_in[idx] + b_in[idx];
36          }
          );
38      }
      );
40
```

Codebeispiel 9.1: oneAPI

In diesem Beispiel umfasst die *Application Scope* die Zeilen eins bis 22. Der Befehl submit von Zeile 24 bis 39 wird als *Command Group Scope* bezeichnet. Dieser enthält von Zeile 32 bis 37 die *Kernel Scope*. Das Ziel ist hier maximales heterogenes Computing und Portierbarkeit des Codes. Der Anwender kann bei der Deviceabfrage zwischen CPU, GPU und FPGA frei wählen. Die Parallelisierung für die entsprechende Architektur geschieht dann automatisch. Es ist geplant, die Intel MKL auch für GPU und Altera FPGAs zu portieren. Wird dann eine API-Funktion für bestimmte Devices aufgerufen, soll automatisch die richtige Architektur in der MKL ausgewählt werden. Es existiert bereits eine Portierung von Tensorflow, um damit maschinelles Lernen mit höheren Python-APIs zu ermöglichen. Kompiliert werden solche Programme mit dem Intel dpc++ Compiler:

```
dpc++ <name>.cpp -lsycl -lOpenCL
```

Der offizielle Release ist erst für 2021 geplant. Jedoch lässt sich eine Betaversion unter https://software.intel.com/en-us/oneapi herunterladen oder sogar mittels Paketmanager installieren. Diese Installation entält die benötigten Compiler, eine OpenCL Version, eine Implementierung von Sycl sowie Profiling Tools, die auch für einfache OpenCL Programme genutzt werden können. Die Betaversion der FPGA-Implementierung kann nur nach Registrierung seperat unter https://software.intel.com/en-us/oneapi/fpga heruntergeladen werden. Eine detailliertere Einführung gibt der *Intel oneAPI Prgramming Guide*. [16]

Grafik-Rendering

10. Grafik-Rendering

10.1. OpenGL

http://www.opengl-tutorial.org/

10.2. Vulkan

https://vulkan-tutorial.com/

10.3. OpenCL Schnittstelle

https://software.intel.com/en-us/articles/
opencl-and-opengl-interoperability-tutorial

10.4. CUDA Schnittstelle

https://www.3dgep.com/opengl-interoperability-with-cuda/

A. Nvidia Jetson

A.1. Produkte

https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/

A.2. Jetpack

https://developer.nvidia.com/embedded/jetpack

A.3. Isaac SDK

https://docs.nvidia.com/isaac/isaac/doc/index.html

B. SPIR-V

https://www.khronos.org/spir/

C. Bildbearbeitung mit OpenCV

https://docs.opencv.org/master/d9/df8/tutorial_root.html

D. DirectCompute

https://www.nvidia.de/object/directcompute_de.html

E. PhysX: Physik Engine für Videospiele

https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxguide/Index.html
https://gameworksdocs.nvidia.com/PhysX/4.1/documentation/physxapi/files/index.html

Literatur

[1] *Accelerating Inference in Tensorflow eith TensorRT (TF- TRT)*. User Guide. Nvidia Corporation. https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html.

[2] *Amdahlsches Gesetz*. https://de.wikipedia.org/wiki/Amdahlsches_Gesetz: de.wikipedia.org.

[3] *AmgX Reference Manual*. Nvidia Corporation. https://github.com/NVIDIA/AMGX/blob/master/doc/AMGX_Reference.pdf.

[4] *CLBLAS Documentation*. Khronos OpenCL Working Group. http://clmathlibraries.github.io/clBLAS/.

[5] *clFFT Documentation*. Khronos OpenCL Working Group. http://clmathlibraries.github.io/clFFT/.

[6] *CUDA Math API*. Nvidia Corporation. https://docs.nvidia.com/cuda/cuda-math-api/index.html.

[7] *CUDA Programming Guide*. Nvidia Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[8] *CUDA Runtime API*. Nvidia Corporation. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html.

[9] *CUDNN Developer Guide*. Nvidia Corporation. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html.

[10] *CUFFT Library User's Guide*. Nvidia Corporation. https://docs.nvidia.com/cuda/cufft/index.html.

[11] *CUSPARSE Library*. Nvidia Corporation. https://docs.nvidia.com/cuda/cusparse/index.html.

[12] *Flynnsche Klassifikation*. https://de.wikipedia.org/wiki/Flynnsche_Klassifikation: de.wikipedia.org.

[13] Matteo Frigo und Steven G. Johnson. *FFTW*. http://www.fftw.org/fftw3.pdf.

[14] Matteo Frigo und Steven G. Johnson. „The Design and Implementation of FFTW3“. In: *Proceedings of the IEEE* 93.2 (2005). Special issue on “Program Generation, Optimization, and Platform Adaptation”, S. 216–231.

[15] Benedict R. Gaster und Lee Howes. *The OpenCL C++ Wrapper API*. Khronos OpenCL Working Group. https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.

[16] *Intel oneAPI Programming Guide (Beta)*. Intel Corporation. https://software.intel.com/sites/default/files/oneAPIProgrammingGuide_7.pdf.

[17] *Keras Guide*. Google. https://www.tensorflow.org/guide/keras.

[18] Dimitar Lukarski und Nico Trost. *Paralution User Manual*. https://www.paralution.com/downloads/paralution-um.pdf.

[19] *Maschinelles Lernen*. https://de.wikipedia.org/wiki/Maschinelles_Lernen: de.wikipedia.org.

[20] *NVCC Reference Guide*. Nvidia Corporation. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[21] *OpenCL 2.0 Reference Guide*. Khronos OpenCL Working Group. https://www.khronos.org/files/opencl20-quick-reference-card.pdf.

[22] *Computer Architectures for Scientific Applications (II)*. 2014.

[23] *Product Selection Guide*. Techn. Ber. https://www.xilinx.com/support/documentation/selection-guides/alveo-product-selection-guide.pdf.

[24] *Programming Guide*. CURAND Library. https://docs.nvidia.com/cuda/curand/.

[25] *Roofline model*. https://en.wikipedia.org/wiki/Roofline_model: en.wikipedia.org.

[26] *Schnelle Fourier-Transformation*. https://de.wikipedia.org/wiki/Schnelle_Fourier-Transformation: de.wikipedia.org.

[27] *SDAccel Environment User Guide*. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf.

[28] *SDx Command and Utility Reference Guide*. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1279-sdx-command-utility-reference-guide.pdf.

[29] *SDx Pragma Reference Guide*. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1253-sdx-pragma-reference.pdf.

[30] *Tensorflow Python API*. Google. https://www.tensorflow.org/api_docs/python/tf.

[31] *TensorRT Developer Guide*. Nvidia Corporation. https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html.

[32] *TensorRT Samples*. Support Guide. Nvidia Corporation. https://docs.nvidia.com/deeplearning/sdk/tensorrt-sample-support-guide/index.html.

[33] *The OpenCL C++ 1.0 Specification*. Khronos OpenCL Working Group. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_Cxx.pdf.

[34] *THRUST Documentation*. Nvidia Corporation. https://thrust.github.io/.

[35] *THRUST Quick Start Guide*. https://docs.nvidia.com/cuda/thrust/index.html.

[36] *Unified Modelling Language*. Object Managment Group. https://www.uml.org/.

[37] *User Guide*. CUBLAS Library. Nvidia Corporation. https://docs.nvidia.com/cuda/cublas/index.html.

[38] *Vivado Design Suite User Guide*. High-Level Synthesis. Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.